Clube Delphi

Edição 162 :: R$ 14,90

DEVMEDIA

Ponteiros, Arrays, Alocação de memória e mais
Explore vários tipos de estrutura de dados em Delphi

Mini curso: Sistema Financeiro – Parte 1
Aplique boas práticas de desenvolvimento

DELPHI XE 7

Fique por dentro das novidades

Incorporando arquivos de recurso
Inclua os mais diversos arquivos em seu executável

Multiprocessamento em Delphi
Entenda como as Threads funcionam e as aplique

MVP

R$ 1.000.000,00
INVESTIDOS EM CONTEÚDO NOS ÚLTIMOS 12 MESES.

APLIQUE ESSE INVESTIMENTO NA SUA CARREIRA...

E MOSTRE AO MERCADO QUANTO VOCÊ VALE!

CONFIRA TODO O MATERIAL QUE VOCÊ TERÁ ACESSO:

- + de **9.000** video-aulas
- + de **290** cursos online
- + de **13.000** artigos
- DEVMEDIA API's consumido + de **500.000** vezes

POR APENAS
R$ 69,90* mensais

*Tempo mínimo de assinatura: 12 meses.

ASSINATURA MVP MOST VALUABLE PROFESSIONAL

PRA QUEM QUER EXIGIR
MAIS DO MERCADO!

DEVMEDIA

Clube Delphi

162ª Edição 2014 ISSN 1517990-7 Impresso no Brasil

Editor Geral

Paulo Quicoli (pauloquicoli@gmail.com)

Equipe Editorial

Guinther Pauli (guintherpauli@gmail.com)

Fabrício Hissao Kawata (fabricio.kawata@bol.com.br)

Giuliano Scombatti Pinto (giuliano@sygnux.com.br)

Daniel Sobrinho Laporte (daniel.laporte@gmail.com)

Jornalista Responsável

Kaline Dolabella - JP24185

Consultor Técnico

Joel Neto (joelrlneto@gmail.com)

Capa

Romulo Araujo

Diagramação

Janete Feitosa

Atendimento ao Leitor

A DevMedia conta com um departamento exclusivo para o atendimento ao leitor. Se você tiver algum problema no recebimento do seu exemplar ou precisar de algum esclarecimento sobre assinaturas, exemplares anteriores, endereço de bancas de jornal, entre outros, entre em contato com:

www.devmedia.com.br/central
(21) 3382-5038

Publicidade

Para informações sobre veiculação de anúncio na revista ou no site e para fechar parcerias ou ações específicas de marketing com a DevMedia, entre em contato com:

publicidade@devmedia.com.br

Distribuição

FC Comercial e Distribuidora S.A
Rua Teodoro da Silva, 907
Grajaú - RJ - 206563-900

Fale com o Editor!

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique a vontade para entrar em contato com os editores e dar a sua sugestão!

Se você estiver interessado em publicar um artigo na revista ou no site ClubeDelphi, entre em contato com os editores, informando o título e mini-resumo do tema que você gostaria de publicar:

Paulo Quicoli - Editor da Revista
pauloquicoli@gmail.com

Paulo Quicoli - **Editor Geral**

pauloquicoli@gmail.com
quicoli.wordpress.com
twitter.com/pauloquicoli

MVP

Assine agora e tenha acesso a todo o conteúdo da DevMedia: **www.devmedia.com.br/mvp**

DEVMEDIA

Sumário

Artigo conteúdo sobre novidades

04 – Novidades do Delphi XE 7
[*Fabricio Hissao Kawata*]

Artigo no estilo série

14 – Estrutura de dados em Delphi – Parte 1
[*Filipe Dalepiane*]

Artigo no estilo curso

24 – Desenvolvendo um Sistema Financeiro em Delphi – Parte 1
[*Filipe Dalepiane*]

33 – Delphi Threads: Utilizando Threads em Delphi
[*Paulo Quicoli*]

44 – Como incorporar arquivos de recursos em executáveis no Delphi
[*Vanderson Cavalcante de Freitas*]

Dê seu feedback sobre esta edição!

A ClubeDelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre esta edição, artigo por artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Novidades do Delphi XE7

Fique por dentro da nova versão

Em tempos recentes, a comunidade Delphi pôde acompanhar de perto um dos períodos de maior evolução do produto, tanto como linguagem quanto como IDE. Se em seu início o mesmo foi concebido como sendo uma ferramenta exclusiva ao desenvolvimento Windows, atualmente seu contexto acaba por englobar diversas outras plataformas. Num ilustrativo inicial, destaque para a inclusão dos recursos que tornaram a ferramenta, no cenário atual, uma ótima opção para o desenvolvimento **nativo** de aplicações móveis, vide dispositivos iOS e Android. Dando um passo seguinte a esta crescente evolução, a mais recente versão lançada do produto, batizada como XE7, acaba por impor novos direcionamentos, bem como a concreta estabilidade de muitos dos recursos recentemente acrescidos ao âmbito da ferramenta. De forma pontual e ilustrativa, a seguir são elencadas algumas das principais mudanças e novidades trazidas neste release.

Fique por Dentro

Recentemente lançado, o Delphi XE7 representa a mais nova versão do principal IDE de desenvolvimento da Embarcadero. Assim como já é tradicional a cada lançamento, o produto traz consigo uma gama de novidades e melhorias que abrangem os diversos e diferentes aspectos que compõem o contexto atual da ferramenta. Conectividade é o termo da vez, numa alusão ao desenvolvimento de aplicações Desktop (Windows e Mac) e Mobile (iOS e Android) que se conectam a dados, serviços na nuvem, dispositivos, sensores, gadgets e outros aplicativos. Por conseguinte, o presente artigo objetiva apresentar uma visão ampla de todo este cenário, pautando no que há de mais novo neste release. Mediante isso, o desenvolvedor estará apto a tirar maior proveito dos recursos pertinentes a esta versão e assim se manter alinhado à própria evolução da ferramenta.

FireMonkey (FireUI)

Até mais recentemente, o FireMonkey ainda era tratado como sendo uma novidade no cenário do desenvolvimento Delphi. Todavia, em decorrência de sua crescente adoção por parte da comunidade, aliado ao seu sucesso produtivo, atualmente o mesmo já pode ser considerado um elemento habitual em todo o processo construtivo que está relacionado ao IDE. Em vista disso, nesta nova versão da ferramenta, a plataforma dá continuidade à sua evolução, porém, sem grandes mudanças significativas mostrando sua estabilidade e maturidade. De início nota-se a introdução de uma nova denominação comercial para a tecnologia, tal qual um codinome ou apelido. Assim, literaturas recentes relacionadas ao XE7 agora podem também se referir ao FireMonkey como FireUI.

Já sob o aspecto do próprio ambiente de desenvolvimento, o primeiro ponto a ser destacado nesta versão envolve a disponibilidade dos tipos de projetos FireMonkey Mobile Application e FireMonkey Desktop Application, até então habituais no release anterior do IDE (Delphi XE6), mas que agora se resumem a um único tipo de projeto, denominado Multi-Device Application (**Figura 1**). Em vista disso, a partir deste ponto o desenvolvedor não necessita mais fazer a escolha inicial explícita entre a criação de uma aplicação Desktop ou

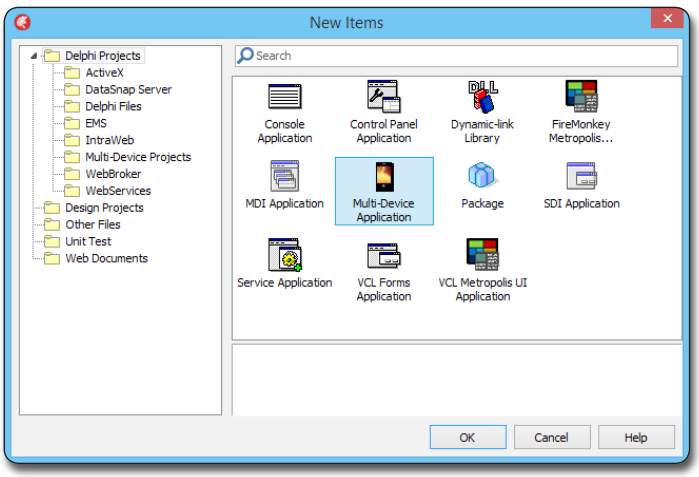

Figura 1. Novo tipo de projeto – Multi-Device Application

Mobile, seguindo agora por uma alternativa multiplataforma única. Posteriormente, a indicação pela plataforma de destino desejada ocorre pelo simples apontamento na opção de Target Platforms do projeto (**Figura 2**).

TMultiView

Todo o poder RAD do Delphi proporcionado por sua ambientação visual incluindo sua extensa gama de componentes já é conhecido do grande público. Dessa forma, a cada novo lançamento novos elementos acabam por ser introduzidos

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

ao contexto da ferramenta, a fim de se manter e elevar toda essa linhagem produtiva. No XE7, a biblioteca FMX ganha um novo componente visual, denominado TMultiView (FMX.MultiView.TMultiView), disponível a todas as plataformas de destino (Target Platforms) previstas para esta segmentação, o que acaba por incluir projetos Desktop (Win32, Win64 e OS X) e Mobile (Android e iOS).

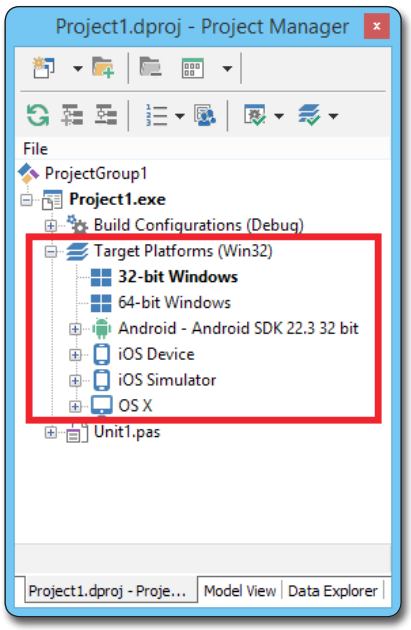

Figura 2. Target Platforms

Conforme sua própria nomeação propõe, sua funcionalidade essencial é atuar como um recipiente para outros controles, tal como um container, fornecendo uma maneira de apresentação sob pontos de vistas alternativos de uma mesma informação. Na elaboração prática, o uso de um TMultiView permite a implementação de uma interface mestre-detalhe, tal como pode ser visto na **Figura 3**. Em termos de especificação, o componente é então dividido em dois painéis representativos:

• Master (Master Pane): este painel tem a função de exibir a coleção de componentes visuais (controles) que irá compor a interface. Em vista disso, é considerado o painel "principal";

• Detail (Detail Pane): painel tido como complementar que exibe informações baseadas nos controles do painel principal.

Save State

Dentre as diversas novidades incorporadas ao FireMonkey neste release está a sua nova capacidade de salvar o estado de uma aplicação antes que ela seja fechada, de modo a recuperá-lo quando a mesma for reiniciada posteriormente. Dada sua caracterização, este recurso, válido tanto para aplicações Desktop quanto para Mobile, é apresentado como Save State e surge como um aliado eficiente ao aspecto multiplataforma que a tecnologia FMX se propõe a atender. Assim como muitas das outras novidades relacionadas ao FireMonkey, na prática, este novo recurso é disponibilizado na classe TForm, o que torna o formulário da aplicação o responsável direto por gerir a persistência e recuperação do estado da mesma. Do ponto de vista técnico, a ação de Save State é direcionada a um evento exclusivo, denominado OnSaveState, enquanto que uma posterior recuperação pode ser elaborada no evento de criação do objeto em questão, tal qual o evento OnCreate do Form. De forma pontual, as informações a serem persistidas pelo Save State incluem aspectos como seleção de controles de marcação (ex: TCheckBox), seleção ativa de um item pertencente a um controle de lista, data, texto, guia ativa (no caso de TTabControl), entre outros.

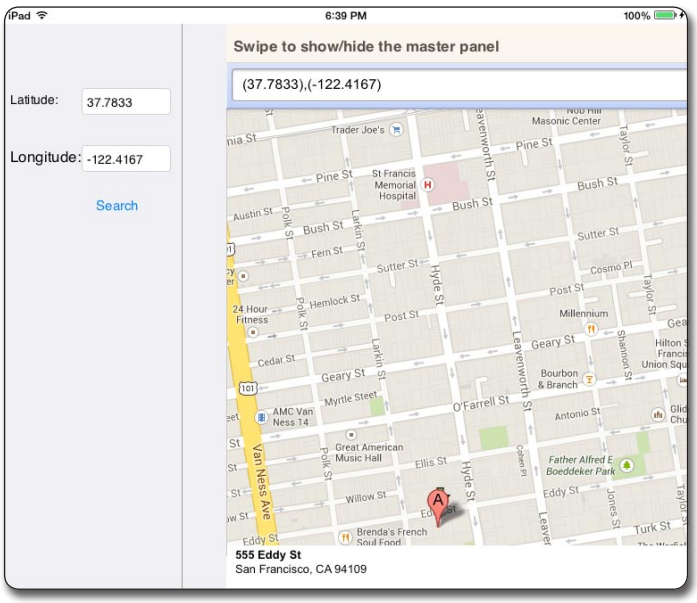

Figura 3. Componente TMultiView em runtime (Fonte: Embarcadero)

Refactoring no framework

Desde seu lançamento, o FireMonkey tem cada vez mais se consolidado como a opção mais determinante, em termos de desenvolvimento, no contexto do Delphi. Isso se justifica pelo fato da tecnologia ser peça fundamental no provimento de aplicações multiplataformas/multidispositivos, o que torna seu uso invariável em construções para Windows, Mac OS, iOS e Android. Diante disso, seu framework tem se mantido em constante evolução, quer seja pela gradual adição de novos elementos e recursos, ou mesmo pela simples adequação da estrutura já existente. Logo, agora no XE7 do ponto de vista estrutural, o FireMonkey apresenta uma série de refatorações internas que acabam por formalizar mais uma vez sua evolução mencionada.

Como exemplo inicial, é possível citar a classe TAnimation anteriormente declarada em FMX.Types, mas que a partir deste release é movida para a unit FMX.Ani. Seguindo por esta mesma linha TNumberBox e TCalendar ganham também novas alocações, mais precisamente units próprias, denominadas FMX.NumberBox e FMX.DateTimeCtrls, respectivamente. Apesar de sutis, todas essas adequações acabam por incidir diretamente em códigos de projetos existentes, fundamentalmente com relação às units declarada na se-

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

ção uses. Adicionalmente, o framework conta ainda com mudanças mais notórias, tal qual a refatoração de métodos. Como mostra disso, StartTriggerAnimation, anteriormente declarado em FMX.Types como parte integrante da classe TFmxObject, agora se apresenta sob o contexto de TAnimator, presente na unit FMX.Ani.

Ainda em termos evolutivos, alguns elementos do framework ganham novos acréscimos, como é o caso da classe TPopup, responsável por prover uma janela pop-up, e que agora conta com três novas propriedades: Size, PopupForm e VisibleStored. Por fim, alguns métodos existentes ganham novas sobrecargas (overloads) a fim de se adequarem de forma efetiva ao recente (e crescente) cenário Mobile da ferramenta, envolvendo o desenvolvimento para Android. Dessa forma, métodos usados com certa precaução em construções de projetos deste tipo, agora ganham novas versões para seu uso sem restrições. Uma prova disso se faz pelo tradicional método MessageDlg, cujas sobrecargas são mostradas a seguir na **Listagem 1**.

Listagem 1. Método MessageDlg

```
function MessageDlg(const AMessage: string; const ADialogType: TMsgDlgType;
const AButtons: TMsgDlgButtons; const AHelpContext: LongInt): Integer;

procedure MessageDlg(const AMessage: string;
const ADialogType: TMsgDlgType; const AButtons: TMsgDlgButtons;
const AHelpContext: LongInt; const ACloseDialogProc: TInputCloseDialogProc);

function MessageDlg(const AMessage: string; const ADialogType: TMsgDlgType;
const AButtons: TMsgDlgButtons; const AHelpContext: LongInt;
const ADefaultButton: TMsgDlgBtn): Integer;

procedure MessageDlg(const AMessage: string;
const ADialogType: TMsgDlgType; const AButtons: TMsgDlgButtons;
const AHelpContext: LongInt; const ADefaultButton: TMsgDlgBtn; const AClose-
DialogProc: TInputCloseDialogProc);
```

Mobile

Sem sombra de dúvidas, o principal atrativo nas recentes versões do Delphi tem sido seu novo suporte ao desenvolvimento de aplicativos para plataformas móveis, mais especificamente iOS e Android. Assim, como não poderia deixar de ser, o Delphi XE7 apresenta uma série de novidades estritamente direcionadas a este contexto, o que inclui desde melhorias a mudanças pontuais.

Nova API para publicidade em aplicações Android

Para a parte de recursos de publicidade (advertising) em aplicações Android, o novo Delphi passa agora a trabalhar com uma nova API – o também novo Google Mobile Ads SDK – que substitui o obsoleto Google AdMob SDK. Assim, a ferramenta acompanha uma imposição de mercado, tendo em vista o anúncio oficial ocorrido no mês de agosto deste ano feito pelo próprio Google, pela descontinuidade da API em questão e, consequentemente, pela não mais aceitação de aplicativos elaborados com a mesma em sua loja oficial de aplicativos (Play Store). Em vista disso, tal atualização diretamente sugere a todos os desenvolvedores uma mudança implícita em suas aplicações Delphi para Android, deste ponto em diante.

Mudanças em TForm e TControl

Com relação a componentes, o XE7 conta com uma série de mudanças, também interpretadas como melhorias, em elementos que condizem estritamente ao cenário de aplicativos touch (de toque), o que acaba por englobar todo o cenário Mobile. Assim, de início, elementos simples como TControl e TForm pertencentes ao framework do FireMonkey, contam agora com um evento denominado OnTap, exclusivo ao toque de usuário no dispositivo e totalmente desvinculado a ação de mouse. Ainda sobre TForm, o mesmo traz ain da outro novo evento relacionado – OnTouch – que lida diretamente com gestos (gestures), novamente sem interpretação de mouse. Além de novos eventos nesta versão, essa mesma classe (TForm) passa a lidar também com o modo immersive full-screen, exclusivo tanto às aplicações móveis para Android 4.4 (KitKat) quanto a aplicativos para Google Glass. Para tal, uma nova propriedade nomeada como FullScreen, é adicionada ao seu contexto. Logo, uma vez marcada como True, o modo imersivo é habilitado.

Seguindo por esta linha a classe TControl, que serve como base para a grande totalidade de controles FMX, ganha uma nova propriedade voltada à gestão e armazenamento do tamanho de um controle. Com base nesta definição, ela é então apresentada sob o nome de Size que, tecnicamente, é declarada sob o tipo TControlSize.

Nova tratativa a métodos de caixa de diálogo

Tradicionalmente, as chamadas para métodos como MessageDlg, ShowMessage, InputBox e InputQuery, também conhecidos como métodos de caixa de diálogo, são consideradas ações de "bloqueio". Isto porque, qualquer código presente após a invocação a qualquer um deles não é executado até que caixa de diálogo exibida seja fechada. Assim, tomando como base a versão anterior imediata do Delphi, a XE6, tal nuance acabava por estabelecer uma restrição de uso em construções móveis, especificamente para Android, uma vez que essa plataforma não permite este tipo de "bloqueio". Mediante isso, agora no XE7, tais métodos ganham um novo parâmetro opcional que, por fim, acaba por viabilizar o uso destes em todas as plataformas previstas ao FireMonkey.

Do ponto de vista técnico, tal parâmetro se apresenta sob o padrão nominal de ACloseXXXProc (Ex: ACloseDialogProc) e permite o provimento de um método anônimo, o qual será chamado imediatamente após a caixa de diálogo ser fechada. Assim, em suma, agora quando da necessidade de execução de algum código após a chamada a um dos métodos citados, este novo parâmetro deverá ser utilizado. Sob a ótica comportamental, com o devido uso deste novo artefato, a chamada a um método MessageDlg, por exemplo, incidirá como forma de "bloqueio" em aplicações Desktop e de "não-bloqueio" em aplicações Mobile. De forma ilustrativa, a seguir na **Listagem 2** são mostradas duas declarações de sobrecarga do método MessageDlg.

FireDAC

Desde a sua oficial incorporação ao cenário nativo do Delphi, o FireDAC vem se mostrando gradativamente mais consolidado ao

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

âmbito da ferramenta a cada novo release lançado. Se inicialmente era disponibilizado apenas como um instalador adicional opcional, na versão anterior do IDE sua evolução deu mostras claras por meio da definitiva incorporação da biblioteca à instalação padrão do Delphi. Em continuidade a este progresso, agora no XE7, diversos aspectos pontuais surgem como adições ao escopo do framework da biblioteca.

Listagem 2. Variantes de MessageDlg

```
procedure MessageDlg(const AMessage: string; const ADialogType: TMsgDlgType;
const AButtons: TMsgDlgButtons; const AHelpContext: LongInt;
const ACloseDialogProc: TInputCloseDialogProc);

procedure MessageDlg(const AMessage: string; const ADialogType: TMsgDlgType;
const AButtons: TMsgDlgButtons; const AHelpContext: LongInt;
const ADefaultButton: TMsgDlgBtn; const ACloseDialogProc: TInputCloseDialogProc);
```

Parâmetros de conexão

Dentre os diversos elementos FireDAC disponibilizados na Tool Palette, o TFDConnection pode ser considerado o principal deles, uma vez que tem a incumbência de efetivamente estabelecer a conexão da aplicação com o banco de dados. Logo, a melhoria inicial apresentada pelo Delphi XE7 refere-se a ele, mais especificamente aos parâmetros de conexão. Pelo lado conceitual nada muda, todavia, a forma de visualização destes no Object Inspector se mostra agora melhorada. Isto porque, até então, por meio da propriedade Params do componente o desenvolvedor tinha acesso a um simples editor textual (**Figura 4**) que pontualmente listava os parâmetros definidos. No XE7 tal propriedade contempla agora um emaranhado de sub-propriedades que possibilitam uma ideal configuração visual. Em suma, cada parâmetro listado no editor anteriormente dá forma agora a um subitem relacionado à propriedade Params, tal como pode ser visto na **Figura 5**.

Melhorias no Data Explorer

Assim como anunciado abertamente em tempos recentes pela Embarcadero, o FireDAC tende a se tornar a principal biblioteca nativa de acesso a banco de dados do Delphi. Em decorrência disso, na mesma época de sua aquisição pela fabricante do IDE, ficou explicitada o "estacionamento evolutivo" do dbExpress, que até então ocupava o posto em questão. Isso fez com que a Embarcadero voltasse seus esforços única e exclusivamente à melhoria e evolução do FireDAC. Logo, se hoje essa alcunha ainda é dividida com o clássico dbExpress, muito provavelmente num futuro próximo sua abrangência deverá ser total. Como prova desta evolução unilateral, a versão XE7 do Delphi conta com melhorias pontuais em sua estrutura de gerenciamento de conexões com banco de dados, o Data Explorer, que estão voltadas exclusivamente ao âmbito do FireDAC.

Apesar de englobar apenas aspectos visuais, as melhorias citadas se apresentam para facilitar o cotidiano profissional do desenvolvedor. Dessa forma, agora toda conexão registrada no

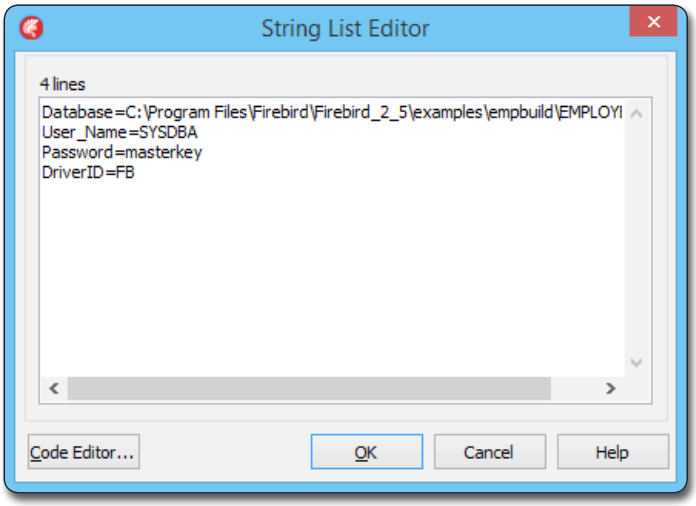

Figura 4. TFDConnection – Editor de parâmetros de conexão

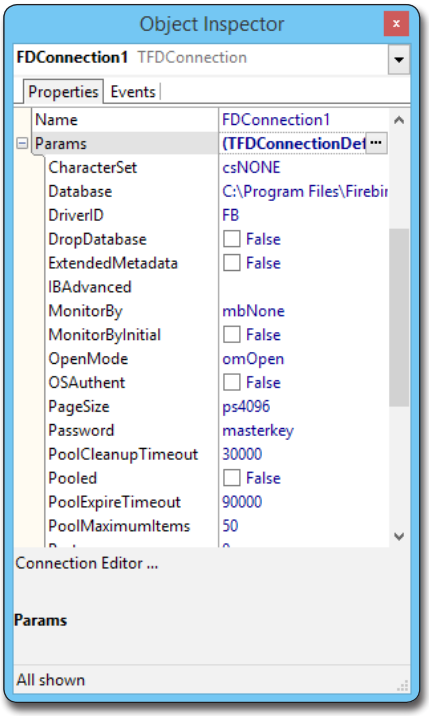

Figura 5. TFDConnection – Sub-propriedades de Params

Data Explorer expõe graficamente representações de três novos tipos de elementos – Primary Key, Foreign Key e Generators/Sequences – que se somam aos já habituais e existentes, vide Tabelas, Campos, Índices, Views e Procedures. Sem deixar de mencionar novamente, vale ressaltar que tais mudanças se aplicam somente ao contexto do FireDAC. De forma ilustrativa, a **Figura 6** mostra uma conexão ao banco Employee do Firebird devidamente registrada na estrutura do Data Explorer, onde podem ser notadas a presença das novas representações dos elementos citados. Em complemento, a **Figura 7** mostra a mesma conexão, sob o âmbito do dbExpress, porém sem a presença dos novos elementos, pautada pelo "estacionamento evolutivo" já citado.

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

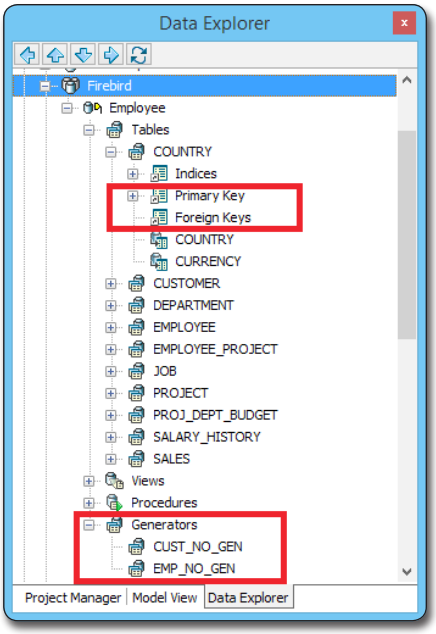

Figura 6. Data Explorer – FireDAC – Novos nós (Primary Key, Foreign Keys e Generators)

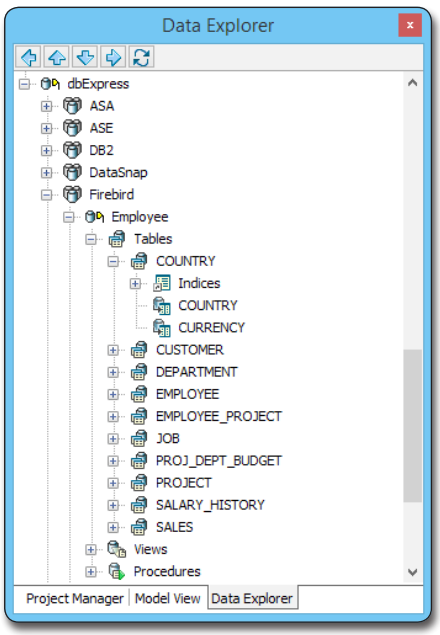

Figura 7. Data Explorer – dbExpress

IDE

Como já é de costume a cada nova versão lançada do Delphi, o XE7 conta também com algumas novidades relacionadas ao ambiente de desenvolvimento propriamente dito. Em meio a pontualidades mais sutis até mais determinantes, é fato que agora todo o cenário habitual apresenta um novo conjunto de diferenças essenciais com relação à versão anterior do produto (XE6).

RAD Studio Guided Tours

Guided Tours é a denominação dada ao novo recurso imposto ao IDE, cujo intuito básico é facilitar o aprendizado sobre a utilização da ferramenta. Assim, o desenvolvedor iniciante tem a oportunidade de conhecer por meio de um passo-a-passo guiado algumas das funcionalidades disponíveis. Na prática, o início de uma "visita guiada" pela ferramenta se dá por meio de um link de ação exclusivo, disponível na Welcome Page (**Figura 8**) do IDE, que por si só já determina o primeiro ponto de interação com o usuário. Por conseguinte, a orientação condizente a cada passo a ser dado pelo visitante se apresenta sob a forma de um balão informativo, contendo o descritivo da ação a ser executada. De forma ilustrativa, a **Figura 9** exemplifica uma dessas mensagens. Aqui, a instrução orienta o usuário a clicar no botão "New Items" para dar início à criação de um novo projeto no IDE.

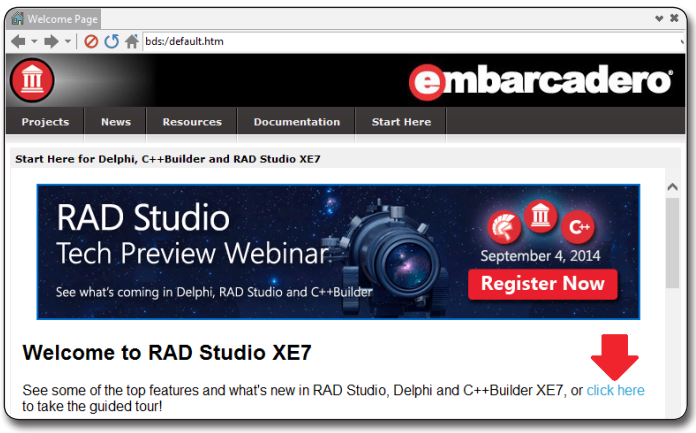

Figura 8. Welcome Page – Guided Tour

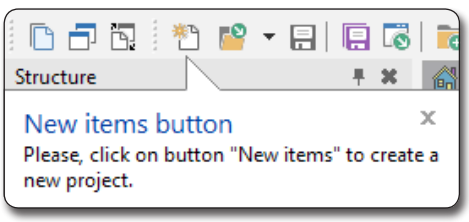

Figura 9. Guided Tour – Balão informativo

Novo Form Designer para aplicações Multi-Device

Com a introdução do novo tipo de projeto para FireMonkey denominado Multi-Device Application, o Form Designer do IDE acaba por ganhar novas características específicas a este cenário. Isto se dá fundamentalmente em razão desta segmentação construtiva atender, a partir de um mesmo ponto em comum, diversas plataformas heterogêneas de destino, onde cada uma dessas determina um visual diferente. Em vista disso, agora no XE7, o designer de formulário do Delphi provê uma série de opções de formatos e layouts para projetos Multi-Device (Desktop e Mobile). Para tal, são disponibilizados dois menus clássicos, do tipo drop-down, denominados Style e Views, que irão prover toda a escolha entre estilizações para Windows, Mac OS X, iOS ou Android. O menu Style tem a incumbência de definir o estilo a ser usado pelo formulário Master (Master Form) da aplicação, que reflete diretamente na estilização dos controles visuais envolvidos,

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

mantendo-se o formato padrão do Form. Em complemento, o menu Views permite uma "estilização" total do formato do formulário, em vista das diversas opções disponíveis (**Figura 10**). Adicionalmente, a disponibilização de componentes na Tool Palette pode ser considerada sensível ao contexto nestes casos, uma vez que sofre alterações mediante o tipo de layout a ser atendido.

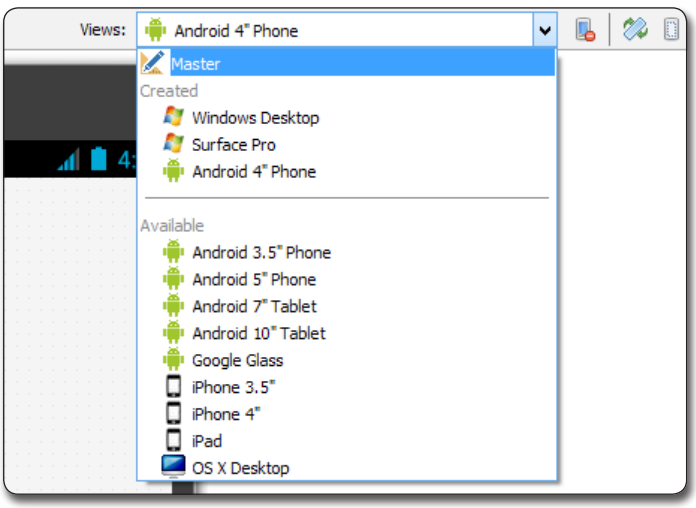

Figura 10. Multi-Device Application – Estilos de layout disponíveis

Adição automática de arquivos de projeto ao Deployment Manager

Já habitual no desenvolvimento móvel, o gerenciador de Deploy (Deployment Manager) provido pelo Delphi pode ser considerado peça fundamental para a ideal distribuição do aplicativo construído. Isto porque, com a dada gama de arquivos relacionados ao projeto – o que inclui arquivos de imagem, arquivos de texto, arquivos multimídia, de configurações, de suporte, bibliotecas, entre outros – torna-se necessária a correta cópia destes ao ambiente de destino, tal qual o do dispositivo físico. Assim, o Deployment Manager é o local onde tudo isso é exposto de maneira clara ao desenvolvedor, auxiliando-o em todo o processo envolvido. No IDE do XE7 uma melhoria sutil, porém significativa, é então acrescida neste ponto: a partir de agora, todo arquivo adicionado ou removido do projeto corrente, por meio do Project Manager, é automaticamente adicionado ou removido do Deployment Manager. Obviamente, no caso específico de arquivos de banco de dados, estes naturalmente acabam por requerer um tratamento especial, uma vez que pode haver arquivos correlatos envolvidos, tais como os relacionados a drivers e licenças. Para estas situações, uma caixa de diálogo apropriada é então apresentada (Featured Files), para que os arquivos complementares necessários possam ser adicionados à lista final de deploy.

Melhorias em App Tethering

Introduzido na versão XE6, o recurso de App Tethering provê a interoperabilidade direta entre aplicações Delphi de diferentes tipos. Num exemplo mais clássico e notório, através de seu uso, é possível estender a amplitude de uma aplicação Desktop, do tipo VCL Forms, de forma que a mesma interaja e compartilhe recursos com uma aplicação FireMonkey Mobile. Em vista das capacidades previstas e da positiva aceitação por parte da comunidade, o Delphi XE7 traz consigo duas novas grandes adições ao recurso.

API for Bluetooth

System.Bluetooth é o nome dado a uma nova unit acrescida à RTL nesta nova versão do IDE. Pelo próprio indicativo, ela então contempla uma API completa para as tratativas de acesso aos recursos desta tecnologia de conexão. Adicionalmente, sua interface de operações é toda voltada ao aspecto multi-device estabelecido pela ferramenta atualmente. Assim, seu uso é irrestrito para as construções de aplicações nativas dos mais variados tipos. Na prática, sua função essencial é permitir que a aplicação seja capaz de acessar os recursos Bluetooth do dispositivo hospedeiro corrente. Por tudo isso, essa novidade poderia ser plenamente categorizada como uma adição "genérica", todavia, muito de seu conceito está intimamente ligado ao recurso de App Tethering. Isto porque, toda a essência de tethering está relacionada à conectividade entre aplicações, o que até então acontecia única e exclusivamente por meio de conexões IP. Agora, diante desta nova API, este cenário evolui, com o surgimento natural de uma nova opção de conectividade, dada por conexões Bluetooth. Sem deixar de mencionar, System.Bluetooth, por meio de sua API, atualmente oferece suporte a duas segmentações da tecnologia: Classic Bluetooth e Bluetooth LE (também conhecido como Bluetooth Low Energy).

Especificação de subnet

Em termos práticos, a partir de uma aplicação tethering, a busca por outros aplicativos condizentes com a tecnologia localizados em dispositivos presentes na mesma rede se dava basicamente pelo uso de métodos – AutoConnect e DiscoverManagers – do componente TTetheringManager. Nos bastidores, a essência de ambos envolve uma atividade que procura por estes devices na sub-rede (subnet) mais direta da rede local (LAN) do aplicativo operante. Esta fundamentação acabava por impor um limite natural ao contexto, impossibilitando a conexão a dispositivos localizados em uma sub-rede diferente ou que possuíssem um endereçamento IP divergente. Em vista disso, agora no XE7 os métodos citados acabam por ganhar uma nova assinatura, com a presença de um novo parâmetro opcional denominado Target, utilizado para se especificar um endereço IP ou subnet, para a busca de dispositivos remotos. De forma a contribuir no entendimento, na **Listagem 3** são mostradas as novas assinaturas citadas:

Programação Paralela

Esta nova versão do Delphi apresenta uma nova biblioteca voltada exclusivamente à programação paralela. Com isso, o desenvolvedor é capaz de aumentar exponencialmente o desempenho do código de suas aplicações, o que vem a resultar numa maior performance da mesma. Na prática, isso é conseguido pela simples inclusão da nova unit System.Threading aos projetos, o que irá proporcionar a utilização de recursos otimizados para o máximo

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

aproveitamento dos recursos de hardware envolvidos, tais como processadores multi-core e memória. Um desses recursos é um novo comando **for** que é de fácil utilização e que por trás utiliza um thread pool que se ajusta automaticamente à carga dos processadores existentes na máquina em execução. A **Listagem 4** mostra o novo comando em uso para calcular um número primo.

Listagem 3. Novas assinaturas

```
//Método AutoConnect
procedure AutoConnect(Timeout: Cardinal; const Target: string = ''); overload;
procedure AutoConnect(const Target: string = ''); overload;

//Método DiscoverManagers
procedure DiscoverManagers(Timeout: Cardinal; const Target: string = ''); overload;
procedure DiscoverManagers(const Target: string = ''); overload;
```

Listagem 4. TParallel For

```
01 TParallel.For(1, 500, procedure (I: Integer)
02 begin
03   if IsPrime (I) then
04    TInterlocked.Increment (Tot);
05 end);
```

A sintaxe pode parecer estranha mas é bem simples. A classe TParallel expõe o método For que recebe como parâmetro dois inteiros, indicando o início e fim do loop e um terceiro parâmetro que nada mais é que um método anônimo que será executado de forma paralela. Nesse exemplo, o método anônimo (**BOX 1**) executa uma função IsPrime (**Listagem 5**) que calcula se um determinado número é primo

Por estar relacionado à RTL, esta nova biblioteca pode ser utilizada para paralelizar toda a gama de aplicações nativas atualmente contempladas pelo IDE, vide construções Windows (32-bit, 64-bit), Mac OS X, iOS e Android.

Listagem 5. Função IsPrime

```
function IsPrime (N: Integer): Boolean;
var
 Test: Integer;
begin
 IsPrime := True;
 for Test := 2 to N - 1 do
  if (N mod Test) = 0 then
  begin
   IsPrime := False;
   break;
  end;
end;
```

Além do for, a unit System.Threading também introduz a class TTask, que representa uma tarefa que deve ser executada de forma paralela. A **Listagem 6** mostra sua utilização.

Foi declarada uma variável do tipo array de ITask, que é a interface implementada por TTask. Utilizamos uma interface aqui por conta de seu recurso de release automático, dessa forma não

BOX 1. Métodos anônimos

Métodos Anônimos podem ser definidos tanto como Procedures quanto Functions, além de poder possuir qualquer número de parâmetros, inclusive nenhum. Podemos usar os já existentes na RTL do Delphi ou criarmos os nossos personalizados. Os tipos de métodos anônimos são declarados como referências para métodos (procedures ou functions). É possível atribuir o método anônimo a variáveis e passá-los também como forma de parâmetro num método. Sua sintaxe é mostrada a seguir:

```
[Nome] = reference to [procedure, function] [Param1: Tipo1, Param2:
 Tipo2, ParamN: TipoN]: [tipo retorno]
```

Resumindo, para declarar um método anônimo precisamos: informar se será um procedimento ou função; especificar o número de parâmetros; informar o tipo de retorno se for função.

A seguir, alguns exemplos.

```
TExemploProcedure = reference to procedure;
TExemploFunction = reference to function: Integer;
TExemploProcedureParam = reference to procedure(x: Integer);
TExemploFunctionParam = reference to function(x: Integer): Integer;
```

Observando a própria RTL do Delphi, é possível notar alguns métodos anônimos. Por exemplo, a unit Sytem.SysUtils define os métodos a seguir:

```
TProc = reference to procedure;
TProc<T> = reference to procedure(Arg1: T);
TFunc<TResult> = referente to function: TResult;
TFunc<T, TResult> = referente to function(Arg1: T): TResult;
```

Listagem 6. Utilizando a classe TTask

```
procedure TFormThreading.Button1Click(Sender: TObject);
var
 tasks: array of ITask;
 value: Integer;
begin
 value := 0;
 tasks := [
 TTask.Create(procedure
  begin
   sleep(1000); // 1 segundo
   TInterlocked.Add(value, 1000);
  end).Start,

 TTask.Create(procedure
  begin
   sleep(3000); // 3 segundos
   TInterlocked.Add(value, 3000);
  end).Start,

 TTask.Create(procedure
  begin
   sleep(5000); // 5 segundos
   TInterlocked.Add(value, 5000);
  end).Start
  ];

 TTask.WaitForAll(tasks);
 ShowMessage('All done: ' + value.ToString);
end;
```


DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

temos que nos preocupar com a liberação de memória de cada objeto TTask instanciado. Posteriormente, adicionamos três instâncias da TTask ao array. O construtor da classe TTask recebe como parâmetro um método anônimo que será executado em paralelo. Como é mostrado, cada método anônimo simplesmente aguarda alguns segundos, utilizando a função sleep, e adiciona um valor à variável value através do método TInterlocked.Add. Esse método cria um ponto de sincronismo que permite a utilização da variável value entre as tasks criadas. Depois que cada task foi criada, foi utilizado o método TTask.WaitForAll para aguardar a finalização das tasks em execução. Se fossemos pensar no tempo que esse código levaria para executar, qual seria nossa resposta? Pois bem, mesmo que temos uma task com pausa de um segundo, outra com pausa de três e a última com pausa de cinco, talvez a resposta pudesse ser nove segundos. Esse resultado seria verdade se não estivéssemos utilizando paralelismo, mas como estamos, o tempo total de execução é de cinco segundos! Ou seja, as tasks com tempo menor foram executadas de forma concorrente à de cinco segundos.

Enterprise Mobility Service

Enterprise Mobility Service, ou simplesmente EMS, é uma solução nativa completa de middleware introduzida na ferramenta nesta nova versão do Delphi. De modo escalável e segura e de características Restful (**BOX 2**) sua estrutura provida pode ser considerada a ideal para o cenário atual de aplicações distribuídas e interligadas, uma vez que provê uma plataforma que pode ser facilmente hospedada tanto localmente quanto em nuvem (Cloud). Esta é então denominada Mobile Enterprise Application Platform e contempla uma API (**BOX 3**) personalizada de recursos, incluindo DataSnap REST e banco de dados. Sua fundamentação é toda baseada em tecnologias abertas e tidas como "padrão", o que inclui aspectos como chamadas de recursos, protocolos de transmissão e formato de dados – REST, HTTP e JSON, respectivamente. Como passo importante para a plena assimilação inicial do assunto, a seguir são listados os principais elementos presentes na arquitetura do Enterprise Mobility Service. Vale destacar também que em algumas fontes de informações espalhadas pela grande rede, tais elementos muitas das vezes são referenciados como componentes, porém, neste caso, numa clara alusão a "componentes de arquitetura" e não a "componentes Delphi".

EMS Server

O servidor EMS é a fundação que provê todo o conjunto de recursos, funções e serviços tidos como essenciais para todo e qualquer cliente (EMS Clients) da plataforma. Além disso, também fica a cargo de sua responsabilidade toda a manutenção do núcleo operacional envolvido, que se refrete na gerência do banco de dados interno (EMS Database) para controle de usuários registrados e suas atividades. Sem deixar de citar, é a partir dessas informações que planos analíticos são montados e disponibilizados no console da solução (EMS Console).

BOX 2. Restful

Restful é o nome de uma atribuição dada a sistemas que seguem os princípios REST, os quais podem ser enumerados em cinco aspectos básicos que envolvem protocolo, operações, sintaxe e hipermídia. O primeiro, protocolo, faz menção aos pedidos e mensagens HTTP. Operações estão relacionadas ao conjunto de métodos HTTP – POST, GET, PUT e DELETE. Sintaxe diz respeito aos recursos que são identificados unicamente por meio de sua URI (Uniform Resource Identifier). Por fim, hipermídia condiz com a representação de estados em sistemas REST, que se dá tipicamente por meio de formatos standard, tais como XML ou HTML.

BOX 3. API

API é o acrônimo para Application Programming Interface, cuja conceituação se faz por um conjunto bem definido de recursos e rotinas estabelecidas, cuja utilização se dá por outros elementos externos, tais como softwares. Por meio de uma interface bem elaborada, a exposição de suas funcionalidades ocorre na forma de serviços, cuja implementação é toda encapsulada dos eventuais consumidores, tornando-os transparentes e de fácil utilização por meio de programação.

EMS Clients

Em suma, os clientes EMS (ou EMS Clients) nada mais são do que as aplicações, móveis ou desktop, que irão de alguma forma acessar os recursos registrados no servidor EMS, usando para isso sua API REST. Sob o âmbito específico dos clientes nativos, neste momento, o EMS provê suporte completo ao FireDAC, o que resulta em um componente específico – TEMSFireDACClient (**Figura 11**). Uma vez formalizadas, essas aplicações são então tratadas como EMS FireDAC Client Applications.

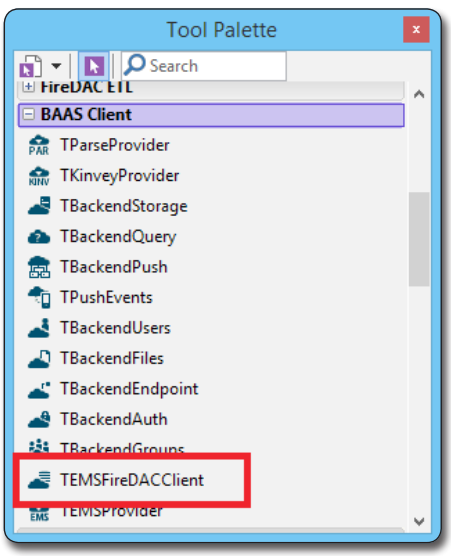

Figura 11. Novo componente TEMSFireDACClient

EMS Console

O console EMS tem a incumbência de ler e interpretar as informações armazenadas no banco de dados da arquitetura a fim de exibi-las no navegador padrão do ambiente. De forma geral, o EMS Console apresenta de forma visual toda a listagem de grupos e usuários registrados, bem como gráficos analíticos relacionados (**Figura 12**).

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Figura 12. EMS Console

API REST

A API REST disponibilizada é utilizada por EMS Clients para acesso às informações do banco de dados da solução através do servidor. Adicionalmente, é possível estender as funcionalidades de um EMS Server existente por meio da criação de packages direcionadas de recursos, aqui tratadas como EMS Packages e, posteriormente, registrando-as no próprio servidor.

Suporte a GIT

Tempos atrás, ainda em sua versão XE, o Delphi apresentou como uma de suas principais novidades a integração de seu IDE com o sistema de controle de versão Subversion. Para a época, tal feito se tornou uma comprovação da busca, por parte da Embarcadero, pela adequação da ferramenta ao âmbito corrente do desenvolvimento de software. Em continuidade a esta evolução, agora no Delphi XE7 são duas as opções, no que diz respeito à integração com sistemas de controle de versão, em decorrência do novo suporte a Git. Por definição, Git é o nome dado a um sistema de controle de versão distribuído e de gerenciamento de código fonte, de caráter livre, cuja essência se assemelha, até certo ponto, ao Subversion. No IDE, toda esta integração prevista se mostra na guia History Manager, que agora dispõe de informações integradas, relacionadas aos dois sistemas de controle de versão em questão.

Como fruto desta nova integração, três ações básicas são atualmente previstas com relação ao versionamento com Git:

- Commit: envio das alterações ao repositório local;
- Clone: cópia (clone) de um repositório remoto para início dos trabalhos locais;
- Log: visualização de log dos projetos versionados.

Vale citar que a plena abordagem sobre o assunto, incluindo os aspectos práticos de utilização, acaba por fugir do escopo do presente artigo. Logo, estes serão tratados em futuros artigos exclusivos sobre o tema.

PAServer Manager

PAServer Manager é o nome dado a um novo aplicativo utilitário direcionado ao desenvolvimento Mac no Delphi. Uma vez que é executado na plataforma de destino, sua funcionalidade principal é prover o gerenciamento de várias instâncias do PAServer, que é o servidor assistente de plataforma atuante neste tipo de desenvolvimento. Aqui, o termo gerenciamento é empregado para designar fundamentalmente cinco ações principais condizentes aos eventuais servidores: parada (stop), reinício (reset), obtenção de informações (get info), edição (edit) e remoção (remove). Numa instalação padrão do XE7, o pacote contendo os elementos envolvidos (PAServer e PAServer Manager), denominado PAServer15.0.pkg é distribuído no seguinte diretório: C:\Program Files (x86)\Embarcadero\Studio\15.0\PAServer

Java2OP

Citando o aspecto do desenvolvimento Android presente até então no contexto do Delphi, o envolvimento de uma biblioteca Java customizada em uma aplicação Mobile acabava por exigir a criação manual de um arquivo "bridge" nativo, referente a um arquivo de classe Java. Em vista desse processo, agora no XE7, um novo utilitário denominado Java2OP (Java to Object Pascal) é disponibilizado ao desenvolvedor. Sua funcionalidade principal é de então automatizar a criação do arquivo citado possibilitando, por exemplo, a geração de um bridge file a partir de um JAR ou arquivos de classes Java. Neste momento, o utilitário não pertence à instalação padrão do produto, sendo disponibilizado apenas para usuários registrados, na central de downloads do IDE na forma de ferramenta auxiliar externa.

BDE

Perante seu contexto, o Delphi XE7 apresenta também algumas subtrações importantes, sendo a principal delas a remoção por completo do BDE. Em termos conceituais, tal tecnologia poderia ser definida como sendo a mais antiga (e obsoleta) solução de banco de dados nativa, que enfim chega a seu ponto final de depreciação nesta versão do IDE. Logo, do ponto de vista prático, sua substituição converge para o uso de tecnologias mais recentes, tais como dbExpress e FireDAC, sendo este último o mais sugerido pela própria Embarcadero.

Perante o âmbito de desenvolvimento que envolve a ferramenta, a total retirada de uma tecnologia, tal como o BDE, tende a impactar sobre diversos aspectos, cujo principal está relacionado à sua ativa utilização em projetos existentes (legados). Isso se justifica pelo fato da tecnologia ainda estar presente em diversas aplicações atuais, atuantes nos mais variados ramos do mercado. Por conseguinte, a fim de se evitar maiores transtornos a esta parcela da comunidade, neste primeiro momento, o BDE ainda será disponibilizado aos usuários registrados, na forma de um instalador externo. Por fim, isso possibilita sua reincorporação e ideal utilização nesta versão da ferramenta. Em decorrência disso, todo o contexto relacionado ao BDE se mantém intacto no Help do IDE.

Mais um ano está se passando e com ele mais uma nova versão do Delphi lançada. Fato é que após um período de críticas e incertezas, a ferramenta tem se mostrado cada vez mais estável,

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

principalmente em razão da estabilização de alguns de seus recursos mais recentes. Como prova maior disso está seu aspecto de construções móveis (Mobile), provido pelo FireMonkey que, desde seu surgimento, ainda na versão XE2, apresenta uma notável crescente, a ponto de hoje ser algo bastante trivial no âmbito do IDE. Nesta mesma toada está o FireDAC, que definitivamente vem para substituir o tradicional dbExpress, no que se refere ao desenvolvimento de aplicações que envolvam banco de dados. Tudo isso é dito a fim de se estabelecer o novo desafio trazido com o lançamento da mais recente versão do Delphi, a XE7, que se faz pela efetiva adoção destes novos elementos em construções correntes e futuras. Assim, se a necessidade pontual do desenvolvedor resulta numa aplicação para Android ou iOS, esta é a hora para a profunda adoção do FireMonkey em suas atividades. Adicionalmente, por se tratar de uma tecnologia multiplataforma, a mesma FMX pode ser usada em substituição à VCL no desenvolvimento Windows. Por fim, o eventual envolvimento de banco de dados a estes cenários proporciona uma abertura para a escolha do FireDAC como principal biblioteca de acesso a dados.

Autor

Fabrício Hissao Kawata
fabricio.kawata@bol.com.br
Formado em Processamento de Dados pela FATEC-TQ e pós-graduado em Engenharia de Componentes. Atua como Analista Programador Delphi há 9 anos. Atualmente escreve artigos sobre Delphi para revistas especializadas. Certified Delphi Developer. Certified Embarcadero Instructor. Site pessoal: www.fabriciokawata.com.br

Links:

RAD Studio XE7 - Página oficial do produto
http://www.embarcadero.com/products/rad-studio

RAD Studio XE7 - Download Trial
https://downloads.embarcadero.com/free/rad_studio

Você gostou deste artigo?

Dê seu voto em ***www.devmedia.com.br/clubedelphi/feedback***

Ajude-nos a manter a qualidade da revista!

Estruturas de Dados em Delphi – Parte 1

Aprenda a utilizar vetores, matrizes, registros e muito mais

Este artigo faz parte de uma série

Um dos segredos de termos softwares rápidos com certeza passa pela correta implementação e utilização de estruturas de dados. A forma como os dados estão organizados durante o processamento fazem com que o sistema tenha uma melhor ou pior performance.

O estudo das mais variadas estruturas de dados costuma ser uma das maiores dificuldades para os desenvolvedores. Observa-se principalmente nas universidades os problemas de muitos alunos de compreenderem esta área da Ciência da Computação que é tão importante no desenvolvimento de softwares.

Nos dias atuais, a maioria das pessoas utiliza a agenda de um telefone celular para fazer o gerenciamento de seus contatos, sejam números de telefones ou e-mails. Este simples recurso de armazenar telefones e e-mails envolve duas tarefas importantes. A primeira é definir como o dado será armazenado na memória, pois é necessário de alguma forma persistir a informação. A segunda tarefa é disponibilizar operações de criação, recuperação, ordenação, alteração e remoção de dados desta agenda.

A primeira tarefa influi muito na performance de qualquer aplicativo, pois quando informações são armazenadas de maneira desorganizada ou incorreta, fica muito mais difícil fazer a manipulação destas informações. Apesar da importância deste armazenamento de informações, o modo como o sistema persiste os dados deve ficar oculto para o usuário final, pois o que realmente interessa para o usuário são maneiras de fazer operações com a agenda e não conhecer a maneira como estes dados são armazenados e recuperados.

Fique por Dentro

O artigo é útil para desenvolvedores que desejam se aprofundar em Estruturas de Dados, gerenciamento de memória, ponteiros e alocação dinâmica, que em geral, são temas de muita importância e muito utilizados no cotidiano nas mais variadas situações. Com conhecimentos avançados nestes temas conseguimos desenvolver formas de armazenamento robustas e que podem ser reaproveitadas em vários projetos de software. Nesta série de artigos o leitor saberá escolher qual a estrutura de dados mais adequada para determinadas situações, além de aprofundar vários tópicos importantes da linguagem Delphi.

Gerenciamento de Memória

Há três áreas de memória onde as aplicações armazenam dados: global memory, heap e stack. Variáveis globais são armazenadas na memória global (global memory). Esta memória é reservada pela aplicação quando o programa inicia e permanece alocada até que o programa termine sua execução. A memória global é também chamada de data segment.

Quando declaramos uma variável dentro de um método, a memória requerida para armazenar tal variável é alocada numa área denominada Stack. Quando o programa sai do escopo do método, as alocações da Stack são liberadas automaticamente.

A memória Stack é alocada dinamicamente utilizando o conceito de pilha (LIFO – Last In First Out). Em Delphi, a memória Stack é utilizada para:

- Variáveis de rotinas locais (métodos, procedimentos e funções);
- Parâmetros de rotinas e tipos de retorno;
- Chamadas a funções da API do Windows;
- Records;

Não é necessário alocar nem liberar a memória Stack, pois este processo é feito automaticamente. O tamanho da memória Stack é, por padrão, grande o suficiente para os programas em Delphi, sendo assim, muito raramente este tamanho terá que ser alterado.

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Nota

Para visualizar o tamanho da Stack, clique no menu Project – Options – Linking e note os campos Maximum Stack Size e Minimum Stack Size. Não recomenda-se que se mude estas configurações, pois isso pode acarretar problemas nas aplicações.

Variáveis locais não são inicializadas quando declaradas, portanto, uma boa prática é antes de utilizar sua variável, inicializá-la para que você não tenha que se deparar com um valor absurdo.

Devido à estrutura de pilha utilizada pela Stack, o acesso a esta memória é muito rápido, pois apenas algumas poucas operações são realizadas para armazenar e ler o valor de uma variável.

Temos ainda a memória Heap, ela é utilizada para:

- Criar uma instância de uma classe;
- Criar e redimensionar arrays dinâmicos;
- Alocar memória explicitamente, usando GetMem, FreeMem, New e Dispose;
- Utilizar Ansi/Wide/Unicode strings, variants e interfaces (gerenciados automaticamente pelo Delphi);

Os blocos de memória alocados na memória Heap não têm uma ordem lógica de acesso, ou seja, seu acesso é randômico. Por isso que operações realizadas na memória Heap são um pouco mais lentas se comparadas às da Stack.

Estruturas de Dados

Em Ciência da Computação as estruturas de dados são definidas como uma forma de armazenamento e organização dos dados, de modo que os mesmos possam ser usados eficientemente.

Existe uma diversidade muito grande de estruturas de dados, cada uma com um leque de aplicações e algumas ainda bem específicas para atender um único domínio de problema.

Quando criamos uma estrutura de dados definimos algumas operações básicas para a mesma:

- Criação da Estrutura;
- Inclusão de Elementos;
- Remoção de Elementos;
- Acesso ao Elemento;
- Etc.

Quando se trabalha com estruturas de dados, geralmente um assunto tratado de forma conjunta é o de análise de algoritmos.

Análise de Algoritmos e Complexidade Computacional

Quando fazemos a análise de algoritmos, o fazemos para provar que o algoritmo desenvolvido está correto e também para determinarmos os recursos computacionais que foram exigidos pelo mesmo, além do tempo de execução, espaço alocado e utilizado, etc.

Comparamos os recursos exigidos por diferentes algoritmos que resolvem o mesmo problema (um algoritmo mais eficiente exige menos recursos para resolver o mesmo problema) e ainda conseguimos prever o crescimento dos recursos exigidos por um algoritmo à medida que o tamanho dos dados de entrada cresce.

A Complexidade Computacional é um ramo da Matemática Computacional (**BOX 1**) que estuda a eficiência dos algoritmos. Para medir a eficiência de um algoritmo frequentemente usamos um tempo teórico que o aplicativo leva para encontrar uma resposta em função dos dados de entrada.

BOX 1. Matemática Computacional

A matemática computacional é uma área da matemática e da computação que colabora na solução de problemas de todas as áreas tanto das ciências exatas, quanto de outras áreas quaisquer, através do desenvolvimento de modelos matemáticos, para o tratamento de problemas complexos, e desenvolvimento de métodos numéricos de obtenção de soluções.

Possui também forte integração com a Modelagem Computacional, uma área de conhecimento que trata da aplicação de modelos matemáticos à análise, compreensão e estudo da fenomenologia de problemas complexos em áreas tão abrangentes quanto as Engenharias, Ciências exatas, Computação, e Ciências humanas.

Atua na interface com outras ciências como a física, a engenharia, a biologia, a ciência da computação, a tecnologia da informação, dentre outras.

O Brasil colabora fortemente com a computação científica com vários centros de estudos avançados.

Tipos Primitivos

O Delphi possui uma série de tipos primitivos, como inteiros, números reais para partes fracionárias, etc.

Os tipos de dados integer são usados para representar números inteiros. Existem vários tipos diferentes de dados integer. Os tipos de dados real são projetados para conter um número com uma parte fracionária.

Na maioria das linguagens, você tem que usar um tipo de dado real para representar um valor monetário. O Delphi fornece o tipo Currency especialmente para este uso. Trata-se de um tipo ponto flutuante que é compatível em relação à atribuição com todos os outros tipos de ponto flutuante. O tipo Currency tem uma precisão de quatro casas decimais e é armazenado como um inteiro de 64 bits (onde os quatro dígitos menos significativos representam os 4 números à direita do ponto decimal). Você pode estar se perguntando por que deveria usar o tipo Currency em vez do tipo de dado real. O tipo Currency oferece duas vantagens importantes:

- Tem uma precisão maior para conter números grandes;
- É usado em CurrencyField e em outros componentes. Ele é compatível com tipos de banco de dados que representam dinheiro.

O tipo de dados Boolean é um dos mais simples e mais usados. As variáveis desse tipo representam um valor lógico: TRUE ou FALSE. As variáveis do tipo boolean aceitam operadores condicionais.

O tipo char foi projetado para armazenar um caracter. Um caracter tem um byte de comprimento, o que daria 256 caracteres que poderiam ser colocados em uma variável do tipo char.

O Delphi permite também o uso de caracteres do sistema UNICODE (Unicode é um padrão que permite aos computadores representar e manipular, de forma consistente, texto de qualquer sistema de escrita existente) utilizando de 2 Bytes.

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

O tipo de dados String tende a ser um pouco mais útil do que o tipo Char. No Delphi, o tipo de dados String era uma concatenação de até 255 caracteres individuais.

A seguir são listados os principais tipos primitivos do Delphi:

• Byte - (Tipo por valor) - Inteiro de 8 bits sem sinal (0 a 255).
• Short - (Tipo por valor) - Inteiro de16 bits com sinal (-32768 a 32767).
• Integer - (Tipo por valor) - Inteiro de 32 bits com sinal (-2147483648 a 2147483647).
• Long - (Tipo por valor) - Inteiro de 64 bits com sinal (-9223372036854775808 a 9223372036854775807).
• Single - (Tipo por valor) - Ponto flutuante 32 bits.
• Double - (Tipo por valor) - Ponto Flutuante 64 bits.
• Decimal - (Tipo por valor) - Ponto Flutuante decimal de 128 bits (1.0 x 10^ -28 a 7.9 x 10^ -28),
• Boolean - (Tipo por valor) - Pode ter os valores True e False.
• Char - (Tipo por valor) - Representa um único caractere unicode de 16 bits.
• Date - (Tipo por valor) - Representa uma data ou hora.
• String - (Tipo por referência) - Representa uma sequência de caracteres unicode.

Recursividade

Muitos problemas têm uma característica comum: cada instância do problema contém uma instância menor do mesmo problema. Dizemos então que esta situação possui uma estrutura recursiva.

A recursividade na prática ocorre quando uma rotina ou método pode invocar a si próprio. Um dos cuidados a ser tomado nesse tipo de implementação é o de cuidarmos para não entrar num loop infinito. Para que isto não ocorra, é necessário que a cada chamada seja alterado o argumento de passagem do método.

Na **Listagem 1** temos um exemplo de emprego do cálculo do fatorial de um número, onde empregamos o cálculo de forma recursiva através da chamada do método Fatorial dentro dele próprio (linha 13).

Listagem 1. Exemplo de Recursividade em Delphi

```
01 program Fatorial;
02
03 {$APPTYPE CONSOLE}
04
05 uses
06   SysUtils;
07
08 function Fatorial(x: Integer): Integer;
09 begin
10   if x = 0 then
11     Result := 1
12   else
13     Result := x * Fatorial(x - 1);
14 end;
15
16 begin
17   Writeln(IntToStr(Fatorial(10)));
18   readln;
19 end.
```

Alocação Dinâmica e Ponteiros

Os ponteiros são um tipo especial de variável que ao invés de guardar um dado, guarda um endereço de memória que referencia um dado, ou seja, elas apontam para determinado endereço. Isto às vezes confunde o desenvolvedor, porque além de saber a informação, é necessário também o conhecimento do endereço daquela informação. Se uma variável do tipo ponteiro apontar para um endereço de memória inválido, recebemos um erro de acesso à memória, o famoso Access Violation.

Quando trabalhamos com variáveis estáticas, o próprio compilador faz o gerenciamento da memória necessária para armazenamento dos dados, não há a necessidade da preocupação do programador no sentido de alocar nem liberar espaço para isso.

Já na alocação dinâmica fazemos a criação e destruição do espaço em memória em tempo de execução, para o armazenamento e processamento da informação. É de responsabilidade do desenvolvedor desalocar todo e qualquer espaço que tenha sido alocado durante a execução do algoritmo.

Ponteiros podem ser declarados de duas maneiras:

• Tipado: ele sabe qual o tipo de informação para a qual está apontando;
• Não-Tipado: ele não sabe para que tipo de dado esteja apontando. Neste caso é necessário fazer uso de typecast para o tipo apropriado, a fim de recuperar a informação.

Na **Listagem 2** temos as mais variadas formas de criação e trabalho com vários tipos de ponteiros. Para declararmos um ponteiro para um tipo específico, basta que informemos um prefixo circunflexo (^) antes do tipo de ponteiro (linhas 13, 15 e 17). Quando desejamos criar um ponteiro para um procedimento, basta que declaremos do tipo procedure e informarmos o número de parâmetros e os tipos deles.

Para fazermos uso do ponteiro para o procedimento, basta atribuirmos o endereço de memória do ponteiro através da arroba (@) e neste instante podemos fazer a chamada do método através do ponteiro, conforme linhas 21 e 22.

Da mesma forma ocorre para acessarmos dados de um valor através de um ponteiro, basta que façamos a atribuição do endereço de memória daquela variável para o ponteiro (linha 25) e já podemos fazer a exibição da informação através do ponteiro. Note que para acessarmos o dado contido no endereço do ponteiro faz-se necessário o uso do circunflexo (^) após o nome do ponteiro.

Para fazermos a criação e liberação de recursos em tempo de execução através de ponteiros, basta que utilizemos a função New para alocarmos recursos e Dispose para liberarmos o recurso da memória após o uso, conforme linhas 35 a 38.

Vetores ou Arrays

O vetor representa a mais primitiva estrutura de dados existente, também pode ser chamado de variável composta homogênea unidimensional, ou seja, uma sequência finita de variáveis alocada dinamicamente e sequencialmente, todas sendo referenciadas com o mesmo nome, e acessadas através de um índice que diferencia

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

cada informação dentro do vetor. Vetores são estruturas de dados lineares compostas de um mesmo tipo de dados (integer, string, record, etc).

Vetor é uma estrutura de dados limitada, pois a quantidade de dados a ser armazenada deve ser predefinida ou redimensionada em tempo de execução possuindo um limite inferior e superior da quantidade de informações que pode ser adicionada.

Existem várias maneiras de criarmos vetores em Delphi, podendo haver muitas personalizações nos tipos de dados suportados nesta estrutura.

Listagem 2. Exemplo de Alocação Dinâmica de Memória e Ponteiros

```
01 program prjPonteiros;
02 {$APPTYPE CONSOLE}
03 uses
04   System.SysUtils;
05
06 procedure Exemplo(Par1: Integer);
07 begin
08   Writeln('Valor recebido: ' + IntToStr(Par1));
09 end;
10
11 var
12   Valor: Integer;
13   PValor: ^Integer;
14   Texto: string;
15   PTexto: ^String;
16   Info: Pointer;
17   Dinamico: ^double;
18   Chamada: procedure(Arg: Integer);
19 begin
20   try
21     Chamada := @Exemplo;
22     Chamada(10);
23
24     Valor := 5;
25     PValor := @Valor;
26     Writeln(PValor^);
27
28     Texto := 'Teste';
29     PTexto := @Texto;
30     Writeln(PTexto^);
31
32     Info := PTexto;
33     Writeln(string(Info^));
34
35     New(Dinamico);
36     Dinamico^ := 15.25;
37     Writeln(FloatToStr(Dinamico^));
38     Dispose(Dinamico);
39   except
40     on E: Exception do
41       Writeln(E.ClassName, ': ', E.Message);
42   end;
43   Readln;
44 end.
```

Vetores estáticos são definidos com tamanho fixo e são imutáveis, que irão perdurar durante todo o ciclo de vida da aplicação sem mudar o número de elementos. Vetores deste tipo podem ter uma ou mais dimensões. Devemos usar arrays estáticos somente quando sabemos a quantidade exata de itens que iremos precisar armazenar no vetor.

Em vetores dinâmicos não são especificados o número de elementos e índice inicial e final, ficando livre para ser definido em tempo de execução através da função SetLength. Este tipo de vetor também pode ter uma ou mais dimensões. Utilizamos arrays dinâmicos quando não sabemos a quantidade de elementos que teremos dentro do vetor, podendo redimensioná-lo em tempo de execução.

A sintaxe de declaração de vetores unidimensionais e multidimensionais pode ser vista na **Listagem 3**, juntamente com a definição de novos tipos de dados baseados em Arrays.

Listagem 3. Sintaxe de um Array

```
01 type Nome: array [Index Type | Ordinal..Ordinal] of Type
02 type Nome: array of Type
03 Nome: array [Ordinal..Ordinal] of Type
04 Nome: array of { array of ...} Type
```

Podemos declarar um novo tipo de array estático (linha 1), onde podemos posteriormente usá-lo para declarar uma nova variável, bastando informar qual será o intervalo de dados. Esse intervalo pode ser definido através de um tipo nativo de dados, como Byte, neste caso o array irá de 0 a 255 que é a dimensão do tipo de dado Byte. Também podemos informar dois valores ordinais, podendo ser eles números (1..10) ou também intervalo de enumerações.

A declaração de um novo tipo dinâmico faz-se unicamente pelo tipo de dado desejado neste vetor.

As declarações de variáveis podem ser estáticas (linha 3) ou dinâmicas (linha 4), sendo as estáticas baseadas em um intervalo de valores ordinais e as dinâmicas simplesmente informando o tipo de dados e especificando quantas dimensões terá este vetor/matriz.

Quando são utilizados os vetores dinâmicos, faz-se necessário fazer uso de uma função chamada SetLength, presente na Unit System, onde especificamos qual vetor desejamos dimensionar e qual será seu tamanho.

Arrays Estáticos

Arrays estáticos são o tipo mais comum de vetores, conforme pode ser visto na **Listagem 4**, onde delimitamos um limite inferior e superior dos índices de nosso vetor (linha 4). Podemos também estabelecer um limite de dados através de um tipo primitivo, como na linha 5, onde definimos que desejamos 256 posições, pois o tipo Byte vai de 0 a 255.

Neste exemplo definimos um vetor estático da posição um até cinco, ou seja, teremos cinco posições para armazenamento de informações. Para conseguirmos acessar todas as informações fazemos uso de um laço de repetição que percorre todos os itens pelo seu índice.

Da mesma maneira que criamos arrays unidimensionais, podemos criar arrays multidimensionais. Arrays de duas dimensões, por exemplo, representam as matrizes, conforme pode ser visto na **Listagem 5**.

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Por ser vetores de duas dimensões precisamos definir dois intervalos de dados, o primeiro é o número de linhas que pode conter a matriz e o segundo é o número de colunas que ela terá (linha 6).

Para fazermos atribuição às posições da matriz faz-se necessário agora o uso de dois índices, onde o primeiro indica a linha e o segundo a coluna que desejamos fazer a atribuição. Para fazermos acesso à informação num laço de repetição, por exemplo, agora necessitamos de dois índices (linha 7) para percorrermos todas as linhas e todas as colunas da matriz.

Listagem 4. Arrays Estáticos

```
01 program prjArrayEstatico;
02 {$APPTYPE CONSOLE}
03 var
04  Nomes: array [1..5] of string;
05  Vetor: array[Byte] of Integer;
06  Ind: Integer;
07 begin
08  Nomes[1] := 'João';
09  Nomes[2] := 'Maria';
10  Nomes[3] := 'Pedro';
11  Nomes[4] := 'Antônio';
12  Nomes[5] := 'Joana';
13  for Ind := 1 to 5 do
14  begin
15   Writeln(Nomes[Ind]);
16  end;
17  Readln;
18 end.
```

Listagem 5. Matriz Estática

```
01 program prjMatrizEstatica;
02 {$APPTYPE CONSOLE}
03 uses
04  System.SysUtils;
05 var
06  Matriz: array [1..2] of array [1..3] of Integer;
07  i, j: Integer;
08 begin
09  Matriz[1, 1] := 10;
10  Matriz[1, 2] := 20;
11  Matriz[1, 3] := 30;
12  Matriz[2, 1] := 40;
13  Matriz[2, 2] := 50;
14  Matriz[2, 3] := 60;
15  for i := 1 to 2 do
16   for j := 1 to 3 do
17    Writeln('Linha ' + IntToStr(i) + ' Coluna ' + IntToStr(j) + ' -' +
18     'Valor: ' + IntToStr(Matriz[i, j]));
19  Readln;
20 end.
```

Podemos ainda ter arrays multidimensionais de mais dimensões, bastando adicionar no final da declaração do array mais um array of e informando mais um índice para atribuição e acesso dos dados.

Arrays de Constantes e Enumerações

Além dos arrays estáticos e dinâmicos podemos ter outros tipos, como o array de constantes e enumerações. Na **Listagem 6** temos o emprego de várias técnicas utilizando estas práticas.

Listagem 6. Exemplo de Arrays de Constantes e Enumerações

```
01 program prjArrayEnum;
02 {$APPTYPE CONSOLE}
03 type
04  TDias = (Segunda, Terça, Quarta, Quinta, Sexta, Sábado);
05  TMeses = array [1..12] of string;
06 const
07  Departamento: array [1..2] of string = ('Financeiro', 'RH');
08  Meses: TMeses = ('Jan','Fev','Mar','Abr','Mai','Jun','Jul','Ago','Set','Out','Nov','Dez');
09 var
10  Dias: array[TDias] of string;
11  DiasUteis: array[Segunda..Sexta] of string;
12  Str, Mes: string;
13 begin
14  DiasUteis[Segunda] := 'Aula Algoritmos';
15 //  DiasUteis[Sábado] := 'Erro de compilação';
16  Dias[Sábado] := 'Folga';
17  Str := Departamento[1];
18  Mes := Meses[3];
19 end.
```

Na linha 4 temos a definição de um tipo que irá aceitar somente dados de dias da semana, já na linha 5, podemos ter mais de uma informação, mas todas elas referentes a meses do ano (de 1 a 12). Na linha 7 temos a definição de uma constante que possibilita o uso de dois departamentos. Já na linha 8 temos a implementação do tipo criado no início para os meses do ano, onde informamos a descrição de cada um deles, além de outros exemplos.

Observe que a Delphi Language possibilita um grande conjunto de declarações, que muitas vezes não fazemos uso no dia-a-dia por falta de conhecimento das mesmas, bastando um estudo mais profundo da linguagem para observarmos quanta flexibilidade ela pode trazer em nossas aplicações.

Arrays Dinâmicos

Arrays dinâmicos são aqueles cujo tamanho não é informado na sua declaração, devendo o desenvolvedor definir essa capacidade em tempo de execução.

Na **Listagem 7** temos a declaração e uso de vetores dinâmicos. A declaração de arrays dinâmicos é semelhante à dos arrays estáticos, apenas não é informado o intervalo de dados que esta estrutura irá suportar (linha 6), devendo o desenvolvedor informar isso em tempo de execução através da função SetLength (linha 3).

Nota

Todos os Arrays dinâmicos definidos através da função SetLength iniciam na posição 0, sendo eles vetores ou matrizes.

Para não termos problemas com o acesso a posições de memória que não existam em um vetor, é recomendado o uso das funções High e Low no for, ou ainda fazermos um for in, conforme pode ser visto nas linhas 19 a 22, onde usamos uma variável auxiliar que irá assumir o dado do vetor na posição atual a cada iteração, ficando o código também bastante elegante.

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Listagem 7. Exemplo de codificação de Arrays Dinâmicos

```
01 program ArraysDinamicos;
02 {$APPTYPE CONSOLE}
03 uses
04  System.SysUtils;
05 var
06  Nomes: array of string;
07  Str: string;
08  i: Integer;
09 begin
10  try
11   System.SetLength(Nomes, 3);
12   Nomes[0] := 'João Augusto';
13   Nomes[1] := 'Pedro Rodrigues';
14   Nomes[2] := 'Almir Arcangelo';
15   for i := low(Nomes) to high(Nomes) do
16   begin
17    Writeln(Nomes[i]);
18   end;
19   for Str in Nomes do
20   begin
21    Writeln(Str);
22   end;
23  except
24   on E: Exception do
25    Writeln(E.ClassName, ': ', E.Message);
26  end;
27  Readln;
28 end.
```

Ordenação de Vetores

Existem situações onde temos um vetor de dados desordenados e desejamos que estejam ordenados de maneira a trabalhar com os mesmos. A maneira mais simples de ordenarmos um vetor é através da comparação e troca dos dados das suas posições, desta forma um laço de repetição é efetuado com as comparações e as trocas. No exemplo da **Listagem 8** será feito uma ordenação em um vetor de dez posições.

Podemos observar no início da listagem o uso da função Randomize (**BOX 2**), logo após através da função Random fazemos a geração de números aleatórios para todas as posições do vetor e mostramos o resultado no próximo laço de repetição (linhas 13 a 17).

A partir daí começamos a fazer a ordenação propriamente dita, onde criamos dois laços de repetição para que possamos fazer a comparação de cada posição do vetor com a próxima e a troca se necessário (linhas 23 a 28).

No final apenas imprimimos novamente o vetor para a comprovação que a ordenação realmente foi efetuada com sucesso.

Classe TArray e método Sort

A classe TArray que está no namespace System.Generics.Collections, é uma classe que não deve ser instanciada, pois possui somente métodos estáticos que possibilitam fazer algumas tarefas em vetores e matrizes genéricas, como buscas e classificações. Podemos ver sua interface na **Listagem 9**.

O primeiro exemplo irá exibir como fazermos a classificação de arrays através do método estático Sort. O mecanismo de ordenação utilizado internamente por este método é o QuickSort (**BOX 3**).

Listagem 8. Algoritmo de Ordenação de Vetores

```
01 program OrdenacaoDeVetores;
02 {$APPTYPE CONSOLE}
03 uses
04  System.SysUtils;
05 var
06  Vetor: array [1..10] of Integer;
07  i, j, aux: Integer;
08 begin
09  try
10   Randomize;
11   for i := Low(Vetor) to High(Vetor) do
12    Vetor[i] := Random(10);
13   for i := Low(Vetor) to High(Vetor) do
14   begin
15    Write(Vetor[i]);
16    Write(';');
17   end;
18   // Ordenação
19   for i := Low(Vetor) to High(Vetor) do
20   begin
21    for j := Low(Vetor) to High(Vetor) do
22    begin
23     if Vetor[j] > Vetor[j + 1] then
24     begin
25      aux := Vetor[j];
26      Vetor[j] := Vetor[j + 1];
27      Vetor[j + 1] := aux;
28     end;
29    end;
30   end;
31   Writeln(#13#10);
32   for i := Low(Vetor) to High(Vetor) do
33   begin
34    Write(Vetor[i]);
35    Write(';');
36   end;
37  except
38   on E: Exception do
39    Writeln('Class: ' + E.ClassName + 'Message: ' + E.Message);
40  end;
41  Readln;
42 end.
```

BOX 2. Randomize e Random

Para efetuarmos a geração de números aleatórios no Delphi, fazemos o uso destas duas funções. A primeira delas, o Randomize serve para fazer a inicialização do gerador de números aleatórios através da hora do dia como referência deste gerador, de forma a criar uma sequência de números totalmente imprevisível.

Já a função Random é a que traz o retorno propriamente dito do valor aleatório. Podemos passar um parâmetro chamado Range, onde definimos qual o limite superior desta geração, indo de zero até o valor informado no parâmetro.

BOX 3. QuickSort

O algoritmo Quicksort é um método de ordenação muito rápido e eficiente, inventado por C.A.R. Hoare em 1960. O Quicksort é um algoritmo de ordenação por comparação não estável.

O Quicksort adota a estratégia de divisão e conquista. A estratégia consiste em rearranjar as chaves de modo que as chaves "menores" precedam as chaves "maiores". Em seguida o Quicksort ordena as duas sublistas de chaves menores e maiores recursivamente até que a lista completa se encontre ordenada.

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Listagem 9. Interface da classe TArray presente em System.Generics.Collections

```
01 TArray = class
02 private
03   class procedure QuickSort<T>(var Values: array of T;
     const Comparer: IComparer<T>;
04    L, R: Integer);
05 public
06   class procedure Sort<T>(var Values: array of T); overload;
07   class procedure Sort<T>(var Values: array of T;
08    const Comparer: IComparer<T>); overload;
09   class procedure Sort<T>(var Values: array of T;
10    const Comparer: IComparer<T>; Index, Count: Integer); overload;
11   class function BinarySearch<T>(const Values: array of T; const Item: T;
12    out FoundIndex: Integer; const Comparer: IComparer<T>;
13    Index, Count: Integer): Boolean; overload;
14   class function BinarySearch<T>(const Values: array of T; const Item: T;
15    out FoundIndex: Integer; const Comparer: IComparer<T>): Boolean; overload;
16   class function BinarySearch<T>(const Values: array of T; const Item: T;
17    out FoundIndex: Integer): Boolean; overload;
18 end;
```

Como nota-se logo na **Listagem 10** através do método QuickSort por exemplo – linha 03, a classe TArray faz uso do recurso de Generics, o que lhe permite ser utilizada com qualquer tipo de dado. (**BOX 4**). Para usar tipos genéricos, devemos fazer a declaração das units Generics.Collections e Generics.Defaults neste exemplo, conforme as linhas 5 e 6 da **Listagem 10**. Declaramos um vetor de strings (linha 8), o qual inicializamos com dez posições (linha 13).

Atribuímos valores aleatórios ao vetor através da função Random em conjunto com a Randomize, que posiciona o gerador de números de acordo com a hora do dia para configurar a geração de números aleatórios com Random.

Ao exibirmos as informações do vetor na tela (linha 17), verificamos que estão totalmente desordenados e com informações que variam de 1 a 10, pois foi o limite que atribuímos para a função Random.

A linha 19 é a que faz o procedimento de ordenação do vetor, por ser de tipo genérico e suportar quaisquer tipos de dados, devemos informar primeiramente que será do tipo String o vetor a ser passado para o método. Além do vetor, informamos qual o mecanismo de ordenação que desejamos, neste caso o a comparação por string através da classe TStringComparer e a opção Ordinal.

Por fim, visualizamos o vetor em tela ordenado (linha 21). Todo o processo foi contornado por um bloco try/except, que possibilita vermos a classe e a mensagem da exceção caso ocorra algum erro durante a execução do algoritmo (linha 24).

BOX 4. Generics

Generics é um dos recursos mais importantes da Orientação a Objetos, pois flexibiliza muito a escrita do código. Como o próprio nome sugere este recurso nos permite trabalhar com uma estrutura genérica, o que faz toda a diferença no caso do Delphi que é uma linguagem fortemente tipada. Quando falamos que uma linguagem é fortemente tipada significa que as estruturas definidas nesta linguagem obrigatoriamente têm que ter um tipo de dado no momento de sua declaração e este tipo de dado não poderá mudar, de uma forma geral seria dizer que o que começa com o tipo inteiro, termina inteiro, por exemplo.

Listagem 10. Ordenando vetor com o método estático Sort

```
01 program ExemploTArray;
02 {$APPTYPE CONSOLE}
03 uses
04   System.SysUtils,
05   Generics.Collections,
06   Generics.Defaults;
07 var
08   Vetor: array of string;
09   i: Integer;
10 begin
11   try
12    Randomize;
13    SetLength(Vetor, 10);
14    for i := 0 to 9 do
15     Vetor[i] := Format('%d', [Random(10)]);
16    for i := 0 to 9 do
17     Write(Vetor[i]);
18    Writeln;
19    TArray.Sort<string>(Vetor, TStringComparer.Ordinal);
20    for i := 0 to 9 do
21     Write(Vetor[i]);
22   except
23    on E: Exception do
24     Writeln(E.ClassName, ': ', E.Message);
25   end;
26   readln;
27 end.
```

Classe TArray e método BinarySearch

A pesquisa ou busca binária é um algoritmo de busca em vetores que também segue o mecanismo de divisão e conquista. Ela parte do pressuposto de que o vetor está ordenado e realiza sucessivas divisões do espaço de busca comparando o elemento buscado (chave) com o elemento no meio do vetor. Se o elemento do meio do vetor for a chave, a busca termina com sucesso. Caso contrário, se o elemento do meio vier antes do elemento buscado, então a busca continua na metade posterior do vetor. E finalmente, se o elemento do meio vier depois da chave, a busca continua na metade anterior do vetor.

A classe TArray do Delphi nos disponibiliza o método BinarySearch que implementa a busca binária e basta que informemos os parâmetros necessários. O método retorna true se encontra o elemento e false se não encontra. Na **Listagem 11** temos um exemplo de uso deste método.

Ao utilizarmos o método estático TArray.BinarySearch (linha 20), passamos o vetor no qual será feita a pesquisa, a variável que irá receber o resultado, caso o elemento procurado seja localizado, e uma variável de saída que irá receber a posição onde foi encontrada a informação.

Registros

Os registros ou records são uma estrutura de dados muito interessante que o Delphi possui. Nela podemos definir um conjunto de campos dentro de um mesmo tipo de dados. Ao contrário de vetores e matrizes, os records podem conter tipos de dados diferentes. Na **Listagem 12** temos a definição da sintaxe de um record.

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Listagem 11. Implementação da Busca Binária através da classe TArray

```
01 program BuscaBinaria;
02 {$APPTYPE CONSOLE}
03 uses
04  System.SysUtils,
05  Generics.Collections,
06  Generics.Defaults;
07 var
08  Vetor: array of string;
09  i, FoundIndex: Integer;
10 begin
11  try
12   Randomize;
13   SetLength(Vetor, 10);
14   for i := 0 to 9 do
15    Vetor[i] := Format('%d', [Random(10)]);
16   for i := 0 to 9 do
17    Write(Vetor[i]);
18   Writeln;
19   i := 3;
20   if TArray.BinarySearch<String>(Vetor, IntToStr(i), FoundIndex,
     TStringComparer.Ordinal) then
21    Writeln(Format('Elemento %d encontrado na lista na posição %d!', [I,
      FoundIndex]));
22   for i := 0 to 9 do
23    Write(Vetor[i]);
24  except
25   on E: Exception do
26    Writeln(E.ClassName, ': ', E.Message);
27  end;
28  Readln;
29 end.
```

Listagem 12. Sintaxe de um Record

```
01 type Nome = record
02  Variaveis...
03 end
```

Da mesma maneira que os vetores e matrizes, necessitamos criar uma variável para representar os records e podemos acessar seus atributos através do ponto. Quando temos muitos campos em um registro, podemos evitar de sempre ter de escrever o nome da variável através do comando With.

Para podermos representar um conjunto de registros da estrutura, podemos utilizar um array de records. Um dos records mais famosos e utilizados no Delphi é o RttiContext da RTTI (**BOX 5**).

Registros Simples

Para trabalharmos com registros em Delphi, precisamos declarar um novo tipo na seção type da unit, conforme **Listagem 13**, linhas 4 a 7. A sua declaração é bastante parecida com a de uma classe, porém neste caso não temos escopos de visibilidade.

Para fazermos uso dos registros, precisamos também de variáveis, como nas declarações das linhas 9 e 10 para que possamos fazer acesso posteriormente no interior do programa. Na linha 9 temos a declaração de dois registros, Tb1 e Tb2, que poderão ser usados depois. Além deles, temos a declaração de um array de registros na linha 10, onde poderemos ter o armazenamento de dez registros.

No caso do uso do vetor, foi necessário que informássemos o índice de qual variável queríamos referenciar naquele momento, conforme linhas 22 e 27.

BOX 5. RTTI

Toda classe deve saber quais são suas propriedades, métodos, campos e atributos. A reflexão, que no Delphi é chamada de RTTI, é um framework interno da ferramenta de desenvolvimento que disponibiliza meios para obtenção de tais informações em tempo de execução.

Através da RTTI podemos obter todos os métodos e propriedades, com seus valores, de uma classe instanciada. Pode ser de qualquer classe, tanto as desenvolvidas por nós, quanto as já existentes no Delphi, como TLabel e TSQLConnection. Através dessas informações, podemos utilizá-las da forma que for necessário, podendo até mesmo alterar seus valores (no caso de propriedades e atributos) usando RTTI.

O uso da RTTI está quase sempre ligado a frameworks que venham a ser desenvolvidos para centralização e otimização dos sistemas. Esses frameworks sendo grandes ou não, têm por objetivo trazer maior flexibilidade a um sistema e produtividade ao desenvolvedor. Temos vários exemplos de utilização como frameworks de validação e persistência de dados.

O Delphi sempre ofereceu recursos de reflexão, porém estes eram limitados quando comparados com a reflexão existente no .NET ou Java. Mas isso até a versão 2009 porque a partir da versão 2010 do compilador, a RTTI passou por uma grande atualização trazendo várias classes novas e benefícios esperados pelos programadores Delphi. As principais classes do novo framework são a TRttiContext, TRttiType, TRttiProperty, TRttiMethod.

Listagem 13. Implementação de Registros

```
01 program prjRegistros;
02 {$APPTYPE CONSOLE}
03 type
04  Tabela = record
05   Nome: string;
06   Chave: string;
07  end;
08 var
09  Tb1, Tb2: Tabela;
10  Tabelas: array[1..10] of Tabela;
11 begin
12  with Tb1 do
13  begin
14   Nome := 'PRODUTOS';
15   Chave := 'ID_PRODUTO';
16  end;
17  with Tb2 do
18  begin
19   Nome := 'FORNECEDORES';
20   Chave := 'ID_FORNECEDOR';
21  end;
22  with Tabelas[1] do
23  begin
24   Nome := 'CLIENTES';
25   Chave := 'ID_CLIENTE';
26  end;
27  with Tabelas[2] do
28  begin
29   Nome := 'FUNCIONARIOS';
30   Chave := 'ID_FUNCIONARIO';
31  end;
32 end.
```

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Registros Avançados

Com o passar das versões do Delphi, foram incorporados muitos recursos nos registros do Delphi, de modo que ele passou a se parecer muito com a definição de uma classe, tendo propriedades, métodos, construtores, propriedades de classes e métodos de classe. O tipo Record é muito utilizado para representar tipos estruturados, que se assemelham a uma classe mas que, na regra de negócio de um sistema, não representam uma entidade. Uma entidade em um sistema orientado a objetos pode conter comportamento, não apenas dados. O foco do Record é estruturar apenas dados, organizando por exemplo o retorno de informações de um método que está sendo criado. Um bom exemplo para isso encontra-se no próprio Delphi ao definir o tipo de TPoint. TPoint nada mais é do que Record que define campos para armazenar valores para as coordenadas X,Y. Não foi necessário criar uma classe, que exigiria sua instanciação no momento do uso, mas a funcionalidade de organizar dados do Record foi suficiente para simplificar passagem ou retorno de parâmetros.

Apesar de agora os registros possuírem muitas novas funcionalidades, os tornando bastante parecidos com classes, existem ainda algumas diferenças importantes entre classes e registros:

- Registros não suportam herança;
- Registros podem possuir tipos variantes, classes não;
- Registros são tipos de valor, diferentemente das classes que são tipos de referências;
- Registros permitem que façamos sobrecarga de operadores, já as classes não permitem que façamos a sobrecarga de operadores;
- Registros são construídos automaticamente utilizando um construtor padrão sem argumentos, já as classes precisam ser criadas explicitamente. Como os registros possuem um construtor padrão, se desejarmos criar um construtor personalizado eles devem obrigatoriamente possuir parâmetros;
- Registros não possuem métodos destrutores, então se desejamos liberar algum recurso da memória devemos criar um método pra isso;
- Registros não podem conter métodos virtuais, dinâmicos ou mensagens;
- Diferentemente de classes, registros não podem implementar interfaces.

Na **Listagem 14** temos a declaração de dois registros. É interessante notar que não se faz necessário criar uma instância de TEndereco para a classe TEmpresa, pois ele é criado automaticamente pelo Delphi. Observe também que a semelhança com as tradicionais classes do Delphi.

Novidades em Arrays no Delphi XE7

A mais nova versão do Delphi, a XE7, recém-lançada pela Embarcadero trouxe novidades no que diz respeito ao trabalho com vetores dinâmicos na linguagem, conforme pode ser visto na **Listagem 15**.

A primeira novidade a se observar é a possibilidade de fazermos a inicialização das informações de um vetor diretamente na sua declaração, conforme linha 5. Agora também podemos fazer a atribuição de várias informações de forma simultânea no vetor, conforme linha 9, além de adicionarmos mais de uma informação ao mesmo tempo (linha 10).

Listagem 14. Utilização de Records avançados

```
01 unit uExemplo;
02 interface
03 type
04  TEndereco = record
05  private
06    FRua: string;
07    FNumero: Integer;
08  public
09    property Rua: string read FRua write FRua;
10    property Numero: Integer read FNumero write FNumero;
11  end;
12  TEmpresa = record
13  private
14    FNome: string;
15    FEndereco: TEndereco;
16  public
17    property Nome: string read FNome write FNome;
18    property Endereco: TEndereco read FEndereco write FEndereco;
19    constructor Create(ANome: string);
20  end;
21 implementation
22 constructor TEmpresa.Create(ANome: string);
23 begin
24  FNome := ANome;
25 end;
```

Listagem 15. Aprimoramentos de Arrays Dinâmicos no Delphi XE7

```
01 program NovidadesArrays;
02 {$APPTYPE CONSOLE}
03 var
04  Vetor1: array of Integer;
05  Vetor2: array of Integer [1,2,3,4,5];
06  Vetor3: array of Integer;
07  Vetor4: array of Integer;
08 begin
09  Vetor1 := [1,2,3,4];
10  Vetor1 := Vetor1 + [5,6];
11
12  Vetor3 := [1,2,3];
13  Insert(5,Vetor3,2);
14
15  Delete(Vetor2, 1, 2);
16
17  Vetor4 := Concat([1,2,3],[4,5,6]);
18 end.
```

Os métodos Insert e Delete agora podem adicionar ou remover informações de qualquer posição do vetor, bastando passarmos corretamente os parâmetros para eles, além da função de concatenação que permite juntarmos dois vetores ou conjuntos de dados e atribuirmos a uma única variável vetor.

As estruturas de dados são um assunto de grande importância no desenvolvimento de sistemas e não é à toa que todos os cursos superiores de graduação na área possuem pelo menos duas disciplinas que abordam profundamente este tema.

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

O que foi visto neste artigo é apenas uma introdução, onde vimos estruturas de dados mais primitivas como arrays, matrizes e registros, que muitas vezes já não são mais utilizadas nos dias de hoje por simplesmente falta de conhecimento.

Nos próximos artigos teremos estruturas de dados mais recentes como Listas, Pilhas, Dicionários, Mapas e várias outras, onde faremos a implementação da estrutura via código para entendermos o funcionamento e armazenamento e também faremos uso de classes que a própria RTL do Delphi nos disponibiliza e que facilitam na hora de precisarmos trabalhar com estruturas de dados.

Autor

FILIPE DALEPIANE
filipe.dalepiane@gmail.com
http://filipedalepiane.blogspot.com.br/
Analista de Sistemas numa Universidade.
Colunista das revistas Clube Delphi e .NET Magazine.
Desenvolvedor Delphi (Desktop) e C# (Web), utilizando bancos de dados SQLServer e Firebird. Bacharel em Ciência da Computação.

Você gostou deste artigo?

Dê seu voto em ***www.devmedia.com.br/clubedelphi/feedback***

Ajude-nos a manter a qualidade da revista!

DÊ UM SALTO EM CONHECIMENTO!

Acesse o maior portal para desenvolvedores da América Latina!

20 mil posts

430 mil cadastrados

10 milhões de page-views por mês

http://www.devmedia.com.br/

DEVMEDIA

Desenvolvendo um Sistema Financeiro em Delphi – Parte 1

Aprenda a desenvolver um sistema financeiro utilizando as melhores práticas

Este artigo faz parte de um curso

Atualmente as empresas necessitam cada vez mais de um número maior de informações, que são geradas rapidamente devido à rapidez com que o mercado se transforma. A velocidade com que as empresas necessitam dessas informações é indispensável para uma boa administração. Sendo que essa velocidade aumenta a cada dia e as empresas que não se adaptarem a essa nova realidade terão dificuldades na obtenção de sucesso e também de permanecer no mercado.

Esta exigência faz com que os controles e relatórios realizados sejam adaptados para que atendam às necessidades essenciais da administração, que é a obtenção de informações necessárias para elaboração de estratégias para sobrevivência e crescimento no mercado.

Portanto, se uma empresa tem como objetivo ampliar seus recursos, terá que elaborar controles dos quais sairão informações para a elaboração de estratégias, que farão com que a empresa consiga se manter atualizada e possa reagir rapidamente frente às mudanças do mercado.

O sistema financeiro de uma empresa consiste em um dos mais importantes setores, pois é vital para a sua sobrevivência. A gestão financeira é uma das mais tradicionais e se encontra em qualquer empresa, seja ela de grande, médio ou pequeno porte, onde são feitas análises e decisões permanentemente, visando sempre maximizar os resultados econômicos.

Fique por Dentro

Um Sistema Financeiro é um módulo com o qual praticamente todo o desenvolvedor Delphi se depara um dia, portanto se torna muito importante entender seu funcionamento e sua importância dentro de uma corporação. Existem muitos tutoriais e artigos explicando maneiras de implementar um sistema deste tipo na web, porém estes na sua totalidade apresentam um desenvolvimento de maneira procedural e baseado em eventos. Neste artigo você terá uma abordagem nova, 100% orientada a objetos e utilizando os principais recursos das versões mais recentes do Delphi.

Para uma empresa manter sua liquidez confortável e obter resultados satisfatórios, há necessidade de tomadas de decisões diariamente. A existência de controles, em especial os financeiros, é essencial para que essas atitudes sejam as mais corretas possíveis.

Pode-se afirmar então que a principal finalidade dos controles financeiros é gerar informações úteis e confiáveis, em tempo hábil, para o empresário tomar as decisões, e para que isso ocorra, os controles gerados devem ser organizados.

Um sistema financeiro de uma empresa consiste basicamente em dois módulos básicos:

- Contas a Pagar;
- Contas a Receber.

Além destes dois módulos básicos, toda empresa precisa de um controle de caixa, onde recursos estão disponíveis para ela a qualquer momento, não precisando que esta espere para que ele se torne disponível.

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

No controle de caixa, onde os operadores ou usuários lançam diariamente todas as operações de entradas e saídas de dinheiro no caixa, para logo após apurar-se o respectivo saldo. Este controle de caixa traz para empresa informações muito importantes como a apuração de vendas e estoque, além dos custos e despesas, possibilitando que se faça análises e se controle todo o dinheiro que entra e sai da empresa.

Contas a Pagar

As contas a pagar são obrigações assumidas pela empresa, que devem ser saldadas dentro do vencimento. É comum nas empresas, que a mercadoria necessária para as operações seja adquirida a prazo, sendo indispensável a devida quitação dos compromissos assumidos dentro dos prazos estabelecidos para evitar transtornos nas próximas compras.

Para organizar todas as contas assumidas, de modo que não fique nenhuma no esquecimento, causando transtornos, deve-se elaborar controles que informem os totais a pagar, obedecendo ao seu vencimento. Desta forma, quando enfrentar dificuldade financeira, a empresa poderá estabelecer prioridades e tentar negociar com os outros credores.

Contas a pagar são contraídas de inúmeras maneiras, como a compra de mercadorias de fornecedores, prestação de serviços de outras empresas para a nossa, além de todas aquelas despesas fixas que são necessárias para manter o funcionamento da empresa, como aluguel, contas de luz, água, telefone, salários, etc.

Relatórios são itens muito importantes no módulo de contas a pagar, pois é possível fazer uma projeção, de forma a antecipar as obrigações da empresa em determinado mês ou período.

Tendo um bom módulo de contas a pagar, conseguimos evitar diversos transtornos, como estes a seguir:

- Prejuízos em decorrência de atrasos no pagamento devido a esquecimentos ou perdas de informações muitas vezes geradas de maneira manual;
- Em caso de dificuldades financeiras, pode-se verificar quais as despesas podem ser negociadas com o credor, de maneira a reduzir juros a serem pagos;
- Gerar informações detalhadas e precisas para fluxos de caixa.

Em nosso sistema de contas a pagar poderemos lançar as obrigações a pagar da empresa e acompanhar a qualquer momento através de relatórios uma lista detalhada de todas estas obrigações em um determinado período de tempo.

Contas a Receber

Não importa o tamanho da empresa, se é pequena, média ou grande, vender a prazo em nosso país é uma grande facilidade e comodidade para os clientes. Em geral, as empresas trabalham com vendas a prazo, sendo esta uma maneira de conquistar os clientes que não têm condições de comprar à vista naquele momento e atualmente é a forma de pagamento mais solicitada.

Mas para que essas vendas a prazo ocorram com segurança, devem ser estudadas maneiras de analisar a concessão de crédito aos clientes, para que o número de inadimplência não se torne muito elevado, e não acarrete para a empresa dificuldades financeiras. Realizada a opção pelas vendas a prazo, a empresa terá que elaborar um controle, de modo que por meio deste ela consiga ter informações necessárias para tomadas de decisões.

Existem diversas maneiras de se fazer uma venda a prazo, sendo algumas delas:

- Cartão de crédito à vista: é onde em geral o recebimento do valor se dará em um período médio de trinta dias, dependendo da data de vencimento do cartão do cliente e a data da compra.
- Cartão de crédito parcelado: é onde o valor total da venda é dividido em várias parcelas iguais e o valor mensal é recebido pela empresa também na data de vencimento do cartão de crédito do cliente.
- Crediário: muito utilizado no comércio varejista em geral, onde geralmente temos o uso de carnês, correspondendo também a uma venda a prazo em parcelas, com ou sem juros. O cliente leva o produto e divide sua dívida em parcelas iguais e paga cada uma mensalmente.
- Boletos bancários: o cliente também pagará parcelado, recebendo boletos bancários com vencimentos mensais que deverá pagar em bancos e estabelecimentos credenciados.

O bom gerenciamento das contas a receber de uma empresa é essencial para a sua saúde financeira. Estes valores a receber na maioria das vezes correspondem ao maior bem da empresa, de forma que deixar de recebê-los ou cobrar indevidamente os clientes, seja no valor, na data, ou na antecipação de um cheque, é muito ruim para empresa e pode manchar seu nome.

De forma bem direta, o controle de contas a receber possibilita:

- Evitar o atraso e inadimplência de pagamentos dos clientes.
- Ter uma previsão exata dos valores que deverão ser recebidos pela empresa em um determinado período de tempo, possibilitando até no caso de dificuldade financeira, vender estas dívidas para instituições bancárias para antecipar seu recebimento.
- Gerar informações completas para o gerenciamento de fluxo de caixa.

O processo que gera uma conta a receber é o de uma venda ou uma prestação de serviço para que se possa gerar uma conta a receber, também pode ser gerada através de uma dívida negociada e renegociada. Desta maneira fazemos a geração destas contas e parcelas a receber no sistema.

Pode-se acompanhar diariamente através de relatórios a situação atual destes títulos, quais estão pagos, quais estão atrasados de maneira bastante profissional.

Uma boa prática que é bastante praticada pelas empresas é o de envio de lembretes aos clientes, através de e-mails, por exemplo, sobre um boleto que está vencendo naquela data ou que está em atraso, podendo ainda propor para os clientes a negociação daquela parcela em atraso. Desta forma evita-se possíveis atrasos dos clientes.

Pode-se ainda gerar gráficos que apresentem demonstrativos de valores que estão para ser recebidos pela empresa de maneira que o administrador financeiro pode analisar as informações.

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

O desenvolvimento

O projeto a ser desenvolvido irá aplicar as melhores práticas de desenvolvimento. Por exemplo, faremos uso de BPLs na prática, algo que por muitos é considerado avançado. Também usaremos a orientação a objetos, aplicando seus conceitos em situações que são comuns no dia a dia de cada desenvolvedor.

Também será bastante utilizado o tratamento de exceções, permitindo que erros sejam manipulados de forma consistente e fazendo com que nossa aplicação possa se recuperar de erros, se possível, ou finalizar sua execução quando necessário, sem perda de dados, perdas de recursos ou ainda vazamentos de memória.

Como sistema gerenciador de banco de dados relacional, faremos o uso do Firebird, na sua versão 3.0, a mais recente versão (**BOX 1**).

Para começar, vamos entender alguns conceitos da orientação a objetos.

BOX 1. Novidades Firebird 3

Na mais recente versão do banco de dados Firebird tivemos a inclusão de alguns interessantes recursos:

• Tipo Boolean: utilizado para armazenamento de valores lógicos True e False, aceitando também valores nulos (null) e desconhecidos (unknown).

• Tipo Identity: define um tipo de dados inteiro que terá seu valor incrementado automaticamente quando este não estiver no comando insert, evitando a necessidade de criação de triggers e stored procedures para este fim.

• Comando Merge: comando utilizado para leitura de informações de uma tabela e inserção/atualização dessas informações em outra através de uma condição.

• Funções Analíticas: são funções de agregação que não restringem o resultado pela cláusula where.

• Expressões Regulares: também chamada de Regex, é uma ferramenta de pesquisa e substituição de textos bastante avançada, composta por caracteres e símbolos.

Encapsulamento

O encapsulamento é algo extremamente importante quando codificamos as classes de um sistema. A ideia é que o sistema seja modular, dividido em várias classes que serão usadas quando necessário para criar instâncias durante a execução do programa para que o mesmo realize as suas funcionalidades.

Quando criamos uma instância de uma classe e chamamos vários métodos para realizar tarefas através de suas ações, não precisamos de fato saber como uma tarefa é realizada, pois está tudo encapsulado na implementação do método. O que queremos é apenas o processamento da tarefa solicitada sem precisar conhecer o funcionamento interno dos métodos da classe instanciada.

O poder do encapsulamento é uma vantagem enorme para o desenvolvimento de software, principalmente para uma softhouse que pode desenvolver componentes de software e utilizá-los em vários projetos ou fornece componentes para terceiros (como é caso de vários pacotes de componentes que podemos instalar nos IDEs modernos, como no próprio Delphi).

Então ter um componente de software com várias funcionalidades encapsuladas sem ser preciso expor seu código fonte ou informar como o mesmo funciona internamente é um grande ganho.

O encapsulamento é a forma elegante de esconder os detalhes da implementação de nossas classes, oferecendo apenas uma interface para acesso. Quando for necessário modificarmos as regras de negócio das nossas classes, elas não irão propagar mudanças em cascata, tornando os softwares muito mais seguros e confiáveis.

Em Delphi temos vários modificadores de acesso e é muito importante que entendamos e saibamos onde aplicar cada um deles no momento certo. São eles:

• Public: modificador que permite acesso aos atributos e métodos de qualquer outra classe;

• Published: semelhante ao public, porém permite acesso através das suas propriedades através do Object Inspector do IDE. Utilizado para a criação de componentes;

• Protected: modificador de acesso que permite acesso aos atributos e métodos nas classes descendentes;

• Private: modificador de acesso que protege totalmente os atributos e métodos, fazendo com que não sejam nunca acessados por outras classes;

• Strict Protected: semelhante ao protected, porém não permite que classes presentes na mesma unit violem o encapsulamento (classes amigas);

• Strict Private: semelhante ao private, porém não permite que classes presentes na mesma unit violem o encapsulamento (classes amigas).

Ao conjunto de propriedades e métodos públicos de nossa classe, chamaremos

Herança

Neste projeto faremos uso intenso de um dos pilares da orientação a objetos, a herança. Porém devemos saber utilizá-la de forma adequada para que não cause um alto acoplamento entre os módulos de nosso sistema.

Herança é um princípio de orientação a objetos, que permite que classes compartilhem atributos e métodos, através de "heranças". Ela é usada na intenção de reaproveitar código ou comportamento generalizado ou especializar operações ou atributos. O conceito de herança de várias classes é conhecido como herança múltipla.

Como exemplo pode-se observar as classes aluno e professor, onde ambas possuem atributos como nome, endereço e telefone. Nesse caso pode-se criar uma nova classe chamada, por exemplo, Pessoa, que contenha as semelhanças entre as duas classes, fazendo com que aluno e professor herdem as características de pessoa, desta maneira pode-se dizer que aluno e professor são subclasses de pessoa.

Packages em Delphi

Os pacotes possuem funcionalidades semelhantes às das DLLs, porém somente podem ser usados em aplicações Delphi, ou seja, uma BPL somente pode ser consumida por sistemas feitos em Delphi, e mais um detalhe, a compilação de uma BPL deve ser feita para cada versão do Delphi, ou seja, uma BPL compilada no

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Delphi XE6 não irá funcionar no Delphi XE7. Já as DLLs permitem, por exemplo, a utilização destas em outras plataformas, como a .NET, permitindo a integração de sistemas.

Existem dois tipos de pacotes:

• Design: utilizado para a instalação de componentes que possam ser adicionados a formulários/data modules em tempo de design;

• Runtime: somente pode ser utilizado em tempo de execução, utilizado para a criação de classes de negócio, DAOs, conexões, etc.

• Design e Runtime: podem conter componentes em tempo de design e classes em tempo de execução.

Em nosso primeiro pacote (File>New>Package Delphi), que chamaremos de Model definiremos que será usado somente em tempo de execução, pois não será adicionado nenhum componente nele, conforme podemos ver na **Figura 1**, onde também configuramos uma descrição para ele.

Desenvolver uma aplicação em pacotes (packages) do Delphi é uma prática pouco comum entre os desenvolvedores, apesar deste recurso já estar presente desde a versão 3 do IDE, mas como veremos no decorrer deste artigo e dos próximos da série, trazem um conjunto muito grande de benefícios e nos fazem codificar de maneira mais elegante e seguindo bons princípios da orientação a objetos.

A maioria dos aplicativos que encontramos na arquitetura Client-Server possuem apenas dois arquivos principais:

• FDB: arquivo de banco de dados (Firebird);

• EXE: arquivo executável do aplicativo.

É muito comum os sistemas apenas com estes dois arquivos, onde o arquivo de banco de dados fica num servidor e o executável roda em cada máquina cliente na rede.

O grande problema desta abordagem é que à medida que o sistema cresce, o executável que representa o aplicativo cliente vai crescendo bastante de tamanho, principalmente se utilizamos bibliotecas de terceiros.

Os pacotes nos possibilitam a modularização dos nossos aplicativos Delphi. Através delas podemos isolar um aplicativo em vários módulos, que facilitam bastante, principalmente na hora de fazermos manutenção e atualização do sistema, pois podemos, por exemplo, atualizar somente um módulo e não todo o sistema. Podemos ainda carregar os pacotes de acordo com a necessidade do usuário, dependendo da funcionalidade solicitada no momento.

Além destas vantagens, o fato de isolarmos os módulos do sistema permite que possamos fazer reuso de partes deste sistema em outros, reaproveitando somente os módulos que nos interessam. Por exemplo, um pacote que contenha as classes de conexão e acesso a dados pode ser reutilizado em outro sistema, fazendo o uso desta estrutura e apenas mudando parâmetros de conexão.

Origens de Pagamento e Recebimento

Tanto no módulo de contas a pagar quanto no módulo de contas a receber temos a figura das origens de pagamentos e recebimentos. Essas origens funcionam como uma categoria para cada lançamento que for efetuado, tanto a receber quanto a pagar e ajudarão muito na hora de buscarmos lançamentos específicos e também na hora de criarmos relatórios.

Para isto iremos criar uma classe que irá representar estas origens no nosso sistema orientado a objetos, chamada TOrigem. O primeiro atributo necessário nesta abordagem é o da descrição, onde será informado um descritivo do que esta informação representa, conforme **Listagem 1**.

Como esta classe servirá tanto para contas a pagar quanto a receber, precisamos de um atributo nela que diferencie a qual tipo de conta cada origem irá se referenciar. Para isso, em sistemas orientados a objetos não criamos flags dos tipos 'P' ou 'R', mas sim criamos um tipo enumerado que irá conter estas duas situações. Neste caso criamos então o enumerado TTipoOrigem, que nos permite informar toPagamento e toRecebimento (linha 6).

Além dos dois atributos privados que irão guardar as informações de descrição e tipo de origem, temos duas propriedades públicas que servirão para que possamos ler e gravar dados nestas propriedades.

Para facilitar a criação de objetos destas classes, além do construtor padrão sem argumentos, temos um construtor que recebe dois parâmetros, a descrição e o tipo de conta, que facilitam na hora de popularmos os atributos da classe, pois ao mesmo tempo que criamos uma instância da classe TOrigem, já fazemos a atribuição das informações da mesma.

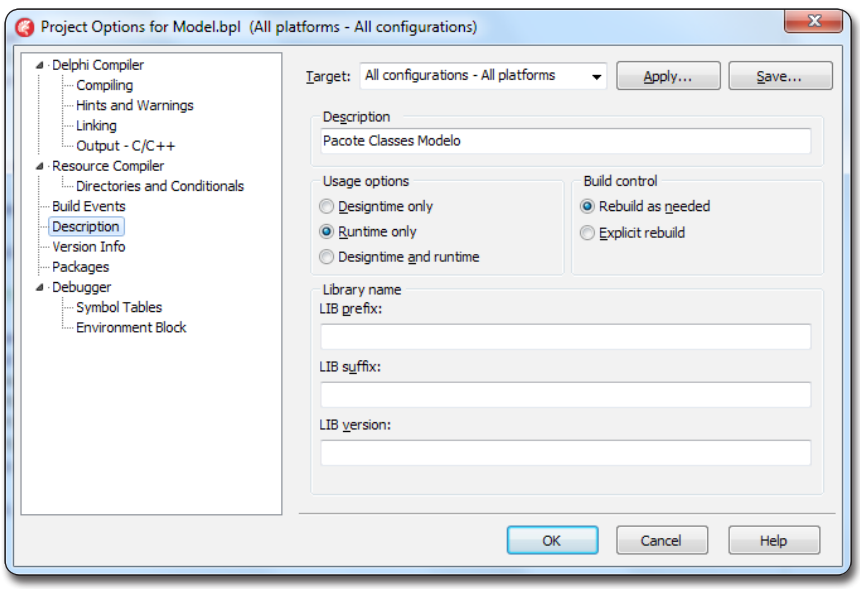

Figura 1. Configuração do Pacote Modelo

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Listagem 1. Enumeração TTipoOrigem e Classe TOrigem

```
01 unit Model.Origem;
02
03 interface
04
05 type
06  TTipoOrigem = (toPagamento, toRecebimento);
07
08  TOrigem = class
09  private
10   FDescricao: string;
11   FTipoOrigem: TTipoOrigem;
12  public
13   property Descricao: string read FDescricao write FDescricao;
14   property TipoOrigem: TTipoOrigem read FTipoOrigem write FTipoOrigem;
15   constructor Create(ADescricao: string; ATipoOrigem: TTipoOrigem);
     overload;
16   constructor Create; overload;
17   function ToString: string; override;
18  end;
19
20 implementation
21
22 { TOrigem }
23
24 constructor TOrigem.Create(ADescricao: string; ATipoOrigem: TTipoOrigem);
25 begin
26  FDescricao := ADescricao;
27  FTipoOrigem := ATipoOrigem;
28  inherited Create();
29 end;
30
31 constructor TOrigem.Create;
32 begin
33  inherited Create();
34 end;
35
36 function TOrigem.ToString: string;
37 var
38  Str: string;
39 begin
40  Str := 'Tipo de Origem: ';
41  case FTipoOrigem of
42   toPagamento: Str := Str + 'Pagamento';
43   toRecebimento: Str := Str + 'Recebimento';
44  end;
45  Str := Str + sLineBreak + 'Descrição: ' + FDescricao;
46  Result := Str;
47 end;
48
49 end.
```

Para que pudéssemos ter dois métodos Create com o mesmo nome, usamos a diretiva overload (**BOX 2**).

O método ToString é definido na classe base de todas as classes do Delphi, a TObject, e por padrão retorna somente o nome da classe. Ele serve para definirmos uma descrição de retorno para cada objeto da classe para que possamos visualizar os dados deste objeto de forma textual.

Por isso sobrescrevemos este método através da diretiva override e implementamos o retorno textual com qual o tipo de origem que se refere aquele objeto, além do valor descritivo do objeto naquele momento, conforme linhas 36 a 47. A constante sLineBreak serve para darmos uma quebra de linha na string.

Note também o nome da unit criada, Model.Origem, devemos sempre fazer o uso de namespace, afim de facilitar o entendimento do sistema e podermos também usar um mesmo nome para vários módulos, por exemplo: Model.Origem, Dao.Origem, View.Origem.

Uma boa prática de programação é testarmos nossas classes de negócio, mesmo que em aplicações console (**Listagem 2**), pois com isso detectamos erros básicos, como referências inválidas, atribuição de campos incorretos, etc.

BOX 2. Overload de Métodos

Overload (ou sobrecarga) de métodos serve para que possamos ter em nossas classes dois ou mais métodos com o mesmo nome. Isso serve também para métodos construtores. Para que o compilador possa diferenciar qual método chamar, os métodos de mesmo nome devem ter um número de parâmetros diferentes ou o mesmo número de parâmetros iguais e apenas os tipos de dados recebidos por ele diferentes.

Devemos adicionar a diretiva overload em cada método. Veja nos exemplos a seguir, uma classe com estes métodos iria compilar sem problemas:

```
procedure Teste(Value: Integer); overload;
procedure Teste(Value: Integer; Str: String) overload;
procedure Teste(Value: String); overload;
```

Nota

O método ToString está disponível somente a partir das versões XE do Delphi e o uso de namespaces somente a partir da versão XE2.

Listagem 2. Teste da Classe TOrigem e Tipo de Origem

```
01 program TestOrigem;
02 {$APPTYPE CONSOLE}
03 uses
04  System.SysUtils,
05  Model.Origem in '..\Model\Model.Origem.pas';
06 var
07  Origem1: TOrigem;
08  Origem2: TOrigem;
09 begin
10  Origem1 := TOrigem.Create('Fornecedores MP', toPagamento);
11  Origem2 := TOrigem.Create();
12  try
13   Origem2.Descricao := 'Clientes Diversos';
14   Origem2.TipoOrigem := toRecebimento;
15   Writeln(Origem1.ToString);
16   Writeln(Origem2.ToString);
17  finally
18   Origem2.Free;
19   Origem1.Free;
20  end;
21  Readln;
22 end.
```

Observe que testamos a criação de instâncias de duas origens, uma de pagamento a outra de recebimento, e já testamos também os dois construtores, além das duas propriedades, para no final exibirmos as informações na tela de console através do método ToString.

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Modelando uma Classe Base Conta

Quando fazemos a concessão de uma venda a prazo para determinado cliente ou fazemos a compra de determinada mercadoria a prazo, temos algumas informações em comum que servirão tanto para o Contas a Pagar quanto do Contas a Receber.

Em nosso modelo proposto, teremos uma classe base chamada TConta (**Listagem 3**) que irá conter todas as informações que são comuns entre as outras duas classes descendentes TContaPagar e TContaReceber, por isso ela será uma classe abstrata, ou seja, não poderá ser instanciada no nosso sistema, pois não faria sentido uma instância dela. Iremos apenas utilizá-la como referência em alguns métodos para aproveitarmos todo o poder que o polimorfismo nos oferece.

Quando pensamos em nossas contas, o primeiro atributo que visualizamos é o valor total dela, onde consta o somatório dos valores de todas as parcelas. Observe que o sistema não irá conter, nesse momento, pagamentos e recebimentos a vista, serão armazenadas somente compras e vendas a prazo. Durante o desenvolvimento do projeto será criada uma classe específica para tratar os recebimentos e pagamentos à vista.

Também teremos um atributo onde ficará armazenada a data de lançamento da conta no sistema, porém a propriedade criada para este campo será somente leitura, pois a ideia é armazenar somente a data de lançamento no sistema e esta será sempre a data atual que será atribuída no método construtor.

Todas as contas também irão possuir uma situação, que indica se ela está em aberto, paga ou foi cancelada. Para isso, mais uma vez fazemos uso de uma enumeração para criarmos estas opções, chamada TSituacaoConta que terá os itens scAberta, scPaga e scCancelada. Este status ficará vinculado através de uma propriedade na classe TConta.

A classe conta também contará com um atributo que irá representar o número total de parcelas contidas. Esta propriedade servirá de auxílio no momento que formos fazer a geração dos itens destas parcelas.

No método de escrita da propriedade ValorTotal (linha 47), veja que temos a verificação se o valor é positivo, a fim de evitarmos a atribuição de um valor errôneo ao atributo. Este é o melhor lugar para fazermos este tipo de validação. Temos também nesta classe um método construtor que irá facilitar nossa vida na hora que precisarmos criar instâncias desta classe.

Veja que a partir da linha 53 temos a implementação do método que adiciona um item que já deve vir criado do módulo cliente e somente o adicionamos na lista, desta maneira conseguimos seguir um princípio da orientação a objetos que indica que devemos ter a inversão de controle e injeção de dependências em nossas classes (**BOX 3**).

Criando os Itens das Parcelas

Todas as contas irão manter uma coleção de itens (parcelas), caracterizando um relacionamento mestre-detalhe.

A classe TItemConta é bastante simples, mas cabem algumas observações e dicas de implementação. Veja na **Listagem 4** que temos três atributos. O número do documento é o atributo que representa um dado de referência para pagamento daquela parcela da dívida a pagar ou receber. Geralmente este número está vinculado com um número de referência de um boleto bancário. O valor obviamente refere-se ao valor a ser pago ou recebido

Listagem 3. Classe base TConta

```
01 unit Model.Conta;
02
03 interface
04
05 uses
06   System.SysUtils, System.Generics.Collections, Model.ItemConta;
07
08 type
09   TSituacaoConta = (scAberta, scPaga, scCancelada);
10
11   TConta = class abstract
12   private
13     FValorTotal: Double;
14     FDataLancamento: TDate;
15     FNumParcelas: Integer;
16     FSituacao: TSituacaoConta;
17     FItensConta: TList<TItemConta>;
18   protected
19     procedure SetValorTotal(const Value: Double); virtual;
20   public
21     property ValorTotal: Double read FValorTotal write SetValorTotal;
22     property DataLancamento: TDate read FDataLancamento;
23     property NumParcelas: Integer read FNumParcelas write FNumParcelas;
24     property Situacao: TSituacaoConta read FSituacao write FSituacao;
25     constructor Create; overload;
26     constructor Create(ANumParcelas: Integer; AValorTotal: Double); overload;
27 public AddParcela(AItemConta: TItemConta);
28   end;
29
30 implementation
31
32 { TConta }
33
34 constructor TConta.Create;
35 begin
36   FDataLancamento := System.SysUtils.Date;
37   FSituacao := scAberta;
38 end;
39
40 constructor TConta.Create(ANumParcelas: Integer; AValorTotal: Double);
41 begin
42   FNumParcelas := ANumParcelas;
43   FValorTotal := AValorTotal;
44   Self.Create();
45 end;
46
47 procedure TConta.SetValorTotal(const Value: Double);
48 begin
49   if FValorTotal > 0 then
50     FValorTotal := Value;
51 end;
52
53 procedure TConta.AddParcela(AItemConta: TItemConta);
54 begin
55   FItensConta.Add(AItemConta);
56 end;
57
58 end.
```

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

daquela parcela. A data de vencimento é o dia em que vence a parcela.

Observe que ainda poderíamos incrementar esta classe, criando atributos para armazenamento de juros no caso de pagamento atrasado ou de desconto no caso de pagamento antecipado da parcela.

BOX 3. Inversão de Controle e Injeção de Dependências

Este princípio de Inversão de Dependência sem dúvida é um dos mais importantes da orientação a objetos, sendo que o mesmo é bastante estudado e existe uma série de frameworks nas mais variadas linguagens de programação para atendê-lo.

O princípio nos ensina que módulos de níveis mais altos do software nunca devem depender de módulos mais baixos, sempre devem depender de abstrações, estas abstrações também não devem depender de detalhes e sim os detalhes das abstrações.

Ao conseguirmos inverter a dependência, fazemos com que as classes clientes não fiquem frágeis em relação a detalhes de implementação, ou seja, se fizermos alguma mudança na implementação nas classes das quais a classe cliente é dependente, não se faz necessário alterações na classe cliente.

Existem diversos padrões de projetos relacionados com este princípio, pois na maioria das vezes desejamos criar interfaces que não dependam de implementações.

Outro padrão que anda junto com a inversão de controle é o da injeção de dependências, pois na realidade a injeção de dependências é uma das maneiras de conseguirmos implementar a inversão de controle. O padrão de injeção de dependências se baseia em um padrão de projeto chamado Builder.

Temos hoje no mercado diversos frameworks de injeção de dependências, principalmente para as linguagens Java e C#, onde o desenvolvimento orientado a objetos é obrigatório. Já no Delphi existem algumas iniciativas como o delphi-spring-framework e o delphiDIcontainer. Ao utilizarmos esses frameworks, as dependências entre as classes não são feitas explicitas programaticamente, e sim na infraestrutura do framework, em uma entidade chamada container, que fica sendo responsável por injetar as dependências da classe no momento apropriado. Neste sentido, as dependências são configuradas através do uso de XML ou através de reflexão computacional com o uso de atributos e anotações.

Existem dois detalhes bastante interessantes no design desta classe que precisam ser mencionados e explicados.

O primeiro refere-se ao fato de não termos nenhuma propriedade que expõe os atributos desta classe, isto porque definimos que estas propriedades serão alimentadas no momento da criação de um objeto da mesma.

O segundo detalhe se refere aos construtores da classe. Veja que temos dois construtores, o primeiro da linha 15 recebe por parâmetro os três parâmetros necessários para popular os atributos. Já o segundo recebe apenas dois parâmetros, data e valor, e no seu interior faz a geração de um número aleatório que irá representar o número do documento conforme linhas 25 e 26.

Agora o interessante fica por conta da linha 27, nela estamos chamando o outro método construtor da própria classe, e por quê? Isso porque este outro método construtor já estava fazendo as operações de atribuição do valor e da data, então porque reescrever este código? Basta chamar o outro construtor passando os parâmetros juntamente com o número de documento gerado. Desta forma economizamos código e evitamos de criar duplicidades de código dentro da classe, facilitando na hora da manutenção.

Listagem 4. Classe que representa os Itens das Contas a Pagar e Receber

```
01 unit Model.ItemConta;
02
03 interface
04
05 uses
06  System.SysUtils;
07
08 type
09  TItemConta = class
10  private
11   FNumDocumento: string;
12   FValor: Double;
13   FDataVencimento: Double;
14  public
15   constructor Create(ANumDocumento: string; AValor: Double;
     ADataVencimento: TDate); overload;
16   constructor Create(AValor: Double; ADataVencimento: TDate); overload;
17  end;
18
19 implementation
20
21 { TItemConta }
22
23 constructor TItemConta.Create(AValor: Double; ADataVencimento: TDate);
24 begin
25  Randomize;
26  FNumDocumento := IntToStr(Random(4));
27  Create(FNumDocumento, AValor, ADataVencimento);
28 end;
29
30 constructor Create(ANumDocumento: string; AValor: Double;
    ADataVencimento: TDate);
31 begin
32  FNumDocumento := ANumDocumento;
33  FValor := AValor;
34  FDataVencimento := ADataVencimento;
35 end;
36
37 end.
```

Clientes e Fornecedores

Em nosso modelo, uma conta a pagar deverá estar vinculada obrigatoriamente a um fornecedor e uma conta a receber vinculada a um cliente. Por esta razão se faz necessário a criação destas duas classes em nosso sistema, conforme **Listagem 5**.

Nota-se que Clientes e Fornecedores compartilham de alguns atributos básicos, como nome e endereço, por esta razão criamos uma classe base chamada TPessoa, da qual TCliente e TFornecedor irão herdar.

Como os clientes, segundo o nosso modelo serão pessoas físicas irão possuir mais uma propriedade chamada CPF e os fornecedores sendo pessoa jurídica possuem o CNPJ, agregando assim estes atributos aos herdados da classe TPessoa.

Observe que neste caso o uso da herança faz todo o sentido e não estamos quebrando a regra do encapsulamento, porque de fato Clientes e Fornecedores são Pessoas, caracterizando perfeitamente a utilização da herança.

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Listagem 5. Classes TCliente e TFuncionario herdando da classe TPessoa

```
01 TPessoa = class
02 private
03   FNome: string;
04   FEndereco: string;
05 public
06   property Nome: string read FNome write FNome;
07   property Endereco: string read FEndereco write FEndereco;
08 end;
09
10 TCliente = class(TPessoa)
11 private
12   FCpf: string;
13 public
14   property Cpf: string read FCpf write FCpf;
15 end;
16
17 TFornecedor = class(TPessoa)
18 private
19   FCnpj: string;
20 public
21   property Cnpj: string read FCnpj write FCnpj;
22 end;
```

Classes de Contas a Pagar e Receber

Na **Listagem 6** podemos verificar as classes TContaPagar e TContaReceber herdando todos os atributos e métodos existentes na classe base TConta.

A classe TContaPagar somente adiciona uma propriedade que referencia a classe TCliente, enquanto que a classe TContaReceber adiciona uma propriedade que referencia a classe TFornecedor.

Cuidados com Encapsulamento de Propriedades

Existe uma má prática de programação bastante comum na orientação a objetos, que é a criação de propriedades e formas de leitura e escrita nessas propriedades, sem uma análise se estas formas de leitura e escrita são realmente necessárias, ferindo o encapsulamento e expondo dados desnecessários de nossas classes.

Tomemos como exemplo da **Listagem 7**, que não tem a ver com o exemplo principal do artigo, mas serve para ilustrar este problema. Imaginemos uma classe TContaBancaria que serve para abstrair uma conta de um cliente em um banco.

Como bem já sabemos e existem muitos tutoriais e artigos sobre este assunto, os atributos devem estar encapsulados na seção private da unit conforme fizemos com o FSaldo na linha 8. Porém este atributo está escondido e queremos fazer acesso a ele. Nesta hora criamos uma propriedade (linha 12) que através de métodos Get e Set (linhas 9 e 10) fazem a leitura e escrita naquele atributo.

Este código é bastante comum, mas pensando bem, qual o real motivo de termos a possibilidade de escrita na propriedade Saldo? O saldo de uma conta nunca deve ser acessado diretamente, devem haver métodos de negócio que irão trabalhar com essa propriedade.

Listagem 6. Classes de Contas a Pagar e Receber

```
01 TContaPagar = class(TConta)
02 private
03   FFornecedor: TFornecedor;
04 public
05   property Fornecedor: TFornecedor read FFornecedor write FFornecedor;
06 end;
07
08 TContaReceber = class(TConta)
09 private
10   FCliente: TCliente;
11 public
12   property Cliente: TCliente read FCliente write FCliente;
13 end;
```

Listagem 7. Criando métodos de leitura e escrita desnecessários

```
01 unit Artigo.Conta;
02
03 interface
04
05 type
06   TContaBancaria = class
07   private
08     FSaldo: Double;
09     function GetSaldo: Double;
10     procedure SetSaldo(const Value: Double);
11   public
12     property Saldo: Double read GetSaldo write SetSaldo;
13   end;
14
15 implementation
16
17 { TContaBancaria }
18
19 function TContaBancaria.GetSaldo: Double;
20 begin
21   Result := FSaldo;
22 end;
23
24 procedure TContaBancaria.SetSaldo(const Value: Double);
25 begin
26   FSaldo := Value;
27 end;
28
29 end.
```

Na **Listagem 8** temos novamente a classe TContaBancaria, mas agora numa versão refatorada, onde tornamos nossa propriedade saldo somente leitura (linha 12), através da eliminação do método de escrita nela, e criamos dois métodos de negócio chamados Deposita e Saca. Estes métodos que irão alterar o valor do atributo FSaldo.

Veja que através do método Saca, garantimos a consistência das informações verificando se o saldo existe para o saque e lançando uma exception criada para este fim.

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Listagem 8. Classe TContaBancaria refatorada com boas práticas

```
01 unit Artigo.Conta
02
03 interface
04
05 uses
06   System.SysUtils;
07
08 type
09   ESaldoInsuficiente = class(Exception);
10
11   TContaBancaria = class
12   private
13     FSaldo: Double;
14     function GetSaldo: Double;
15   public
16     property Saldo: Double read GetSaldo;
17     procedure Deposita(AValor: Double);
18     procedure Saca(AValor: Double);
19   end;
20
21 implementation
22
23 { TContaBancaria }
24
25 procedure TContaBancaria.Deposita(AValor: Double);
26 begin
27   FSaldo := FSaldo + AValor;
28 end;
29
30 function TContaBancaria.GetSaldo: Double;
31 begin
32   Result := FSaldo;
33 end;
34
35 procedure TContaBancaria.Saca(AValor: Double);
36 begin
37   if FSaldo >= AValor then
38     FSaldo := FSaldo - AValor
39   else raise ESaldoInsuficiente.Create('Saldo Insuficiente!');
40 end;
41
42 end.
```

Portanto fica esta dica para quem quer seguir fielmente as regras de encapsulamento. Nunca declare propriedades com métodos de leitura e escrita que não sejam realmente necessários. Isso serve também para outros métodos públicos. A interface da nossa classe deve ser a mais enxuta possível, com apenas aquilo que realmente deve ficar exposto para as classes que precisarão consumi-la, desta forma terão sistemas mais confiáveis e que podem ser ditos orientados a objetos, seguindo boas práticas de programação.

O desenvolvimento utilizando pacotes em Delphi ajuda muito na hora da manutenção, é bastante recomentado o seu uso por parte dos desenvolvedores. Neste primeiro artigo da série vimos apenas a criação do pacote que referencia a camada Model do nosso sistema financeiro, a camada onde ficam os objetos PODO. PODO é a sigla para Plain Old Delphi Objects, que nada mais são do que objetos puramente Delphi, que não possuem dependência de frameworks externos.

Autor

FILIPE DALEPIANE
filipe.dalepiane@gmail.com
http://filipedalepiane.blogspot.com.br/
Analista de Sistemas numa Universidade.
Colunista das revistas Clube Delphi e .NET Magazine.
Desenvolvedor Delphi (Desktop) e C# (Web), utilizando bancos de dados SQLServer e Firebird. Bacharel em Ciência da Computação.

Você gostou deste artigo?

Dê seu voto em ***www.devmedia.com.br/clubedelphi/feedback***

Ajude-nos a manter a qualidade da revista!

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Delphi Threads: Utilizando threads em Delphi

Conheça e aplique em seu dia a dia

Para podermos fazer uso efetivo de threads e da programação multithread em si, é preciso compreender primeiramente o que é: processo, módulo e thread.

• **Processo:** Podemos dizer de forma simplista que um processo é um programa que é executado em uma máquina, compartilhando seu processamento e memória. Partindo para uma definição mais técnica, sabemos que um processo executa um bloco de comandos, e esse processo o executa de forma isolada. Os recursos como memória, disco e processamento utilizados são virtualizados de tal forma que cada processo possui seu próprio conjunto de recursos, que não é acessível a outros processos.

• **Módulo:** Os processos executam módulos. Esses podem ser totalmente diferentes, por exemplo, um mesmo processo pode executar um código de envio de e-mail enquanto que outro pode executar um download de página Web. Toda a execução desses módulos acontece de forma simultânea. Por exemplo, ao pensarmos no sistema operacional Windows, o mesmo não deixa de realizar uma operação de I/O (código de arquivos) enquanto você imprime um documento no Word.

• **Thread:** As threads foram desenvolvidas quando ficou claro que o tempo de processamento poderia ser melhor utilizado. Em outras palavras, quando se desejava que um sistema executasse várias ações, de forma simultânea. Um bom exemplo desse tipo de ação é a verificação de grafia enquanto está se digitando um texto. O processo de entrada não é parado esperando pelo processo de correção. Os dois são executados simultaneamente. Hoje é muito comum máquinas com mais de um núcleo de processamento, porém sem dúvida alguma o número desses processadores físicos é menor que o número de threads que podem rodar em paralelo. Para contornar isso, os sistemas operacionais oferecem suporte ao time slicing. O time slicing nada mais é do que o controle de execução das threads, onde cada uma é executada por um curto período e essa execução é alternada como em um rodízio. Isso permite que um único processador execute várias threads.

Fique por Dentro

Neste artigo aprenderemos como utilizar threads em Delphi. Veremos alguns dos conceitos teóricos suficientes para aplicar de forma adequada esse recurso pouco utilizado no desenvolvimento diário. Apresentamos um problema simples, abordando nele situações reais como, por onde começar, como cancelar uma thread, como impedir que um sistema seja fechado se suas threads ainda estão em execução para posteriormente abordar seu uso em conjunto com banco de dados. Será utilizada a classe TThread, disponível antes mesmo do Delphi 7, porém os exemplos são construídos utilizando Delphi XE 6.

Quando as threads são úteis?

As threads são úteis quando um aplicativo necessita realizar uma operação longa. Mas como identificar esse processamento? Vamos imaginar que um sistema efetua uma consulta de clientes e no momento da pesquisa o usuário desse sistema começa a redigir um e-mail, enquanto o sistema continua efetuando a pesquisa, porém agora a janela desse sistema avisa que o "sistema parou de responder". O usuário leigo vai simplesmente pensar que o sistema travou e irá fechá-lo. Porém, como programadores, sabemos que o sistema em si não estava travado. A consulta poderia estar muito bem sendo realizada, contudo, devido ao longo e pesado processamento, o processo principal do sistema não pôde mais responder às mensagens do sistema operacional e assim, o sistema operacional acha que o sistema está congelado. Colocar esse

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

processamento pesado encapsulado em uma thread faria com que o sistema, que roda sob um processo principal, continuasse respondendo ao sistema operacional. Ou seja, uma thread não altera o propósito de um sistema, ela simplesmente muda o tempo de execução das operações, tornando-se uma solução elegante para esse tipo de situação.

Outra situação onde o uso de thread é útil é quando temos disponível um ambiente multiprocessado, que como já foi dito, é muito comum. Assim, podemos fazer com que um aplicativo desenvolvido por nós, possa fazer uso eficiente de mais de um núcleo de processamento. O uso consciente das threads pode fazer com que esse sistema, que algumas vezes parece que está lento, se torne mais responsivo.

Entendendo como um programa funciona

Para facilitar o entendimento sobre threads, vamos imaginar um programa VCL bem simples. A **Figura 1** mostra o funcionamento de um formulário principal com um único botão que ao ser clicado executa uma chamada ao comando ShowMessage.

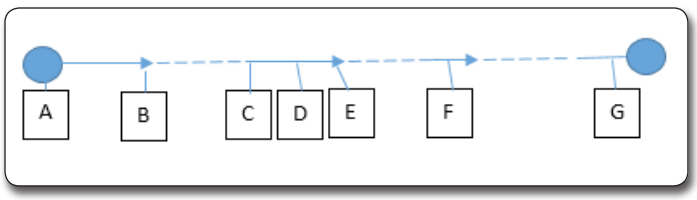

Figura 1. Processamento de uma aplicação simples

No ponto A o programa é iniciado e até chegar no ponto B, a thread do aplicativo – que é a thread VCL – espera por mensagens do sistema operacional. Se se nada é recebido ou enviado, até chegar no ponto C a thread é então suspensa. No ponto C é onde o usuário clica no botão, que leva a execução ao ponto D, onde a mensagem é exibida. Uma vez exibida chegamos no ponto E, onde o aplicativo aguarda pelo "OK" na tela. Enquanto não se pressiona o OK, a thread fica em suspensão. Ao clicar no OK (ponto F) a mensagem é fechada e a thread entra novamente em suspensão até o usuário finalizar o aplicativo no ponto G.

Alguns pontos dessa visualização precisam ser destacados. Um deles é que a thread não é mantida em execução continuadamente. Existem períodos onde ela não recebe estímulo externo e não está processando nada enquanto isso. Os recursos alocados pela aplicação estão lá, a janela é visível mas nenhum de seus códigos é executado. Quando a aplicação é iniciada, a thread principal é iniciada criando a janela principal. Ao finalizar esse seu propósito, o código recai em uma parte da VCL que executa um loop contínuo que requisita ao sistema operacional por mais mensagens. Se não há nada a ser processado, o próprio sistema operacional suspende a thread. Mas, quando o usuário clica no botão da janela, a thread é acordada novamente, recebendo a mensagem que houve o clique no botão. Essa operação de suspensão e reativação acontece várias vezes.

Cálculo de número primos

Identificar se um número é primo ou não pode ser uma operação custosa de acordo com o tamanho do número a ser calculado. Essa é uma operação que pode ser executada em uma Thread, liberando a thread principal para outras atividades. O Delphi encapsula uma thread através da classe TThread, que traz em sua implementação várias peculiaridades já resolvidas. Para experimentar seu uso vamos criar um novo projeto do tipo VCL Forms Application no Delphi XE6. Após salvar o projeto como DevMediaPrimos, adicionamos uma nova classe que representa a thread de cálculo. Isso é feito pelo menu File>New>Other, que abre uma janela para escolhermos o que adicionar. Entre as opções exibidas temos justamente Thread Object. Vamos escolher essa opção, que então nos solicita o nome da classe. Para esse exemplo daremos o nome de CalculoPrimosThread. A unit criada é então salva como CalculoPrimosThreadU. Essa opção criou uma nova classe que herda de TThread. Nessa classe vamos adicionar uma seção public e nela declararemos uma nova propriedade Numero, do tipo integer. Já em sua seção protected acrescentaremos uma função de nome EhPrimo, que retorna um boolean. Essa é a função que realiza o cálculo em si. Como é possível observar, já existe o método Execute. Esse método irá por sua vez executar a função EhPrimo. Na **Listagem 1** temos a implementação da função EhPrimo. Não entramos em detalhes da execução do cálculo pois não é objetivo do artigo.

Listagem 1. Cálculo de números primos

```
1. function CalculoPrimosThread.EhPrimo: boolean;
2. var
3.  iter: integer;
4. begin
5.  result := true;
6.  if FNumero < 0 then
7.  begin
8.   result := false;
9.   exit;
10. end;
11. if FNumero <= 2 then
12.  exit;
13. for iter := 2 to FNumero - 1 do
14. begin
15.  if (FNumero mod iter) = 0 then
16.  begin
17.   result := false;
18.   {exit;}
19.  end;
20. end;
21. end;
```

A **Listagem 2** apresenta o método Execute, que faz uso da nossa função EhPrimo.

O formulário principal é bem simples, como mostra **Figura 2**. No evento OnClick do botão, o processo é o de instanciar e executar a thread CalculoPrimosThread, como mostra a **Listagem 3.**

Nota

É preciso adicionar na cláusula uses a unit CalculoPrimosThreadU.

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Listagem 2. Método Execute

```
1. procedure CalculoPrimosThread.Execute;
2. begin
3.  if EhPrimo then
4.   ShowMessage(IntToStr(FNumero) + ' é primo.')
5.  else
6.   ShowMessage(IntToStr(FNumero) + ' não é primo.');
7. end;
```

Listagem 3. Executando o cálculo

```
1. procedure TForm1.btnCalcularClick(Sender: TObject);
2. var
3.  NovaThread: CalculoPrimosThread;
4. begin
5.  NovaThread := CalculoPrimosThread.Create(True);
6.  NovaThread.FreeOnTerminate := True;
7.  try
8.   NovaThread.Numero := StrToInt(editNumero.Text);
9.   NovaThread.Resume;
10.  except on EConvertError do
11.   begin
12.    NovaThread.Free;
13.    ShowMessage('Não é um número válido!');
14.   end;
15.  end;
16. end;
```

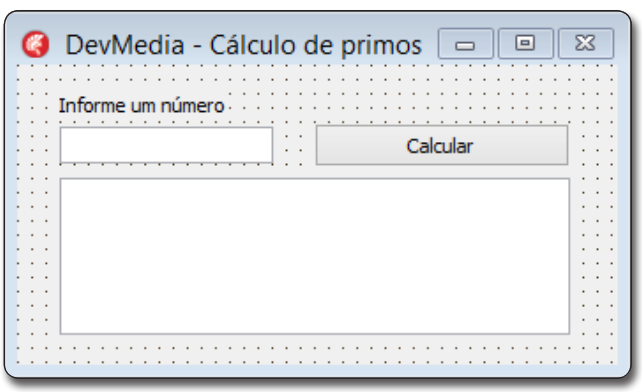

Figura 2. Cálculo de primos

Temos uma variável do tipo CalculoPrimosThread que é instanciada. Observamos que em seu construtor passamos o valor True, indicando que a thread irá iniciar de forma suspensa, ou seja, não será executada nesse momento. Isso permite a passagem de parâmetros, como FreeOnTerminate, que irá limpar os recursos alocados assim que o processamento terminar, e Numero, que é o número que será verificado se é primo ou não.

Para testar seu funcionamento, podemos adicionar um número bem grande, algo como "1234567891" e clicar no botão calcular. A thread principal não ficará presa, pelo contrário, enquanto o primeiro cálculo é e realizado, podemos facilmente inserir um valor bem menor, por exemplo "2" e mandar calcular. A resposta será imediata, mesmo enquanto o primeiro cálculo ainda é realizado. Isso só é possível porque a cada clique estamos instanciando uma nova thread e solicitando sua execução.

Contudo, algo não ficou claro. O que aconteceria se o resultado devesse ser armazenado em um objeto ou o que aconteceria se o usuário fechasse o aplicativo enquanto uma thread ainda não finalizasse seu processamento?

Compartilhamento de dados e sincronização

Uma determinada informação pode ser compartilhada entre diversas threads. Por exemplo, uma variável pública pode ser acessada por diversas threads. Aparentemente não há nada demais nisso, porém os problemas começam quando essas threads alteram o valor desse recurso compartilhado, afetando operações que dependem dessa variável. Isso acontece porque existe um agendador no sistema operacional que controla a execução das threads, podendo parar uma e executar outra a qualquer momento. Para garantir a integridade de um dado, a linguagem Delphi implementa o método TThread.Syncronize.

Esse método é bem simples, recebendo como parâmetro um outro método sem parâmetros que contém o que deseja ser executado. Vamos modificar o aplicativo de demonstração. Para começar, nosso objeto TThread irá não mais exibir uma caixa de diálogo, mas irá preencher um TMemo localizado no formulário principal. Vamos adicionar então um componente TMemo no formulário e chamá-lo de MemoResposta. Então vamos modificar a thread para acessar esse TMemo.

Em CalculoPrimosThread adicionamos um outro método chamado AtualizarResultados, que acessa o formulário passando o resultado. Ao invés de chamar ShowMessage, chamamos Synchronize, passando como parâmetro o método AtualizarResultados. A **Listagem 4** mostra o código da classe CalculoPrimosThread.

Listagem 4. CalculoPrimosThread atualizada

```
1.  CalculoPrimosThread = class(TThread)
2.  private
3.   FNumero: integer;
4.   FResultado: string;
5.   procedure SetNumero(const Value: integer);
6.   { Private declarations }
7.  protected
8.   function EhPrimo: boolean;
9.   procedure AtualizarResultados;
10.  procedure Execute; override;
11. public
12.  property Numero: integer read FNumero write SetNumero;
13. end;
14. ...
15. procedure CalculoPrimosThread.AtualizarResultados;
16. begin
17.  Form1.memoResposta.Lines.Add(FResultado);
18. end;
19. ...
20. procedure CalculoPrimosThread.Execute;
21. begin
22.  if EhPrimo then
23.   FResultado := IntToStr(FNumero) + ' é primo.'
24.  else
25.   FResultado := IntToStr(FNumero) + ' não é primo.';
26.  Synchronize(AtualizarResultados);
27. end;
```

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

A linha 17 mostra que a Thread está acessando o formulário principal, mas isso só é possível por conta da utilização do método Synchronize (linha 26). Para testar, novamente inserimos um longo número para cálculo e na sequência um número pequeno. O formulário principal é atualizado de forma adequada conforme as threads vão sendo processadas.

Quando o método Synchronize é chamado, a thread de cálculo de números primos é pausada. Nesse momento então a thread principal, VCL, também entra em suspensão, permitindo a execução do loop de mensagens (já mencionado anteriormente). Quando a thread da VCL entra nesse estado, o método que foi passado como parâmetro em Synchronize é executado então na thread da VCL, garantindo que não ocorram conflitos na thread da VCL, porque ela vai executar o método como se estivesse respondendo uma mensagem do Windows.

Controlando o término de threads

Quando o usuário encerra um aplicativo principal que está executando várias threads, esse término da thread principal força o encerramento de todas threads secundárias sem elas terem encerrado suas operações, o que pode levar a consequências não pensadas. Digamos que uma determinada thread está gravando um arquivo, e o usuário simplesmente encerra o programa principal, forçando o encerramento da gravação desse arquivo. Qual a situação desse arquivo com o término abrupto da thread? Ou, os recursos alocados pela thread foram devidamente liberados?

Para contornar essa situação, o Delphi oferece um mecanismo onde é possível fazer com que uma thread notifique outra, avisando que deve ser encerrada. A classe TThread possui o método Terminate e a propriedade Terminated, que possui o valor padrão false quando o objeto é criado. Sempre que o método Terminate for executado, a propriedade Terminated será modificada para True, cabendo a todas as threads verificarem se elas precisam ser canceladas adequadamente. A propriedade Terminated funciona apenas como um flag, ela não garante que outras threads que checaram que Terminated está marcado para True vão iniciar seu processo de finalização. A **Listagem 5** exibe o método EhPrimo modificado para verificar o término de execução.

A linha 14 mostra o uso da propriedade Terminated. Caso seja True, o loop é interrompid, podendo com isso retornar um resultado incorreto, mas para o exemplo proposto é aceitável.

Uma vez que uma thread finaliza sua execução, o evento OnTerminate é disparado. Por ocorrer no contexto da VCL, assim como os métodos executados pelo comando Synchronize, pode ser de grande utilidade, permitindo a criação de um mecanismo que permita a uma aplicação somente ser finalizada se suas threads forem terminadas. A ideia é simples: possuir um contador que é incrementado quando uma thread é criada e decrementado quando alguma thread é finalizada. Então, ao fechar o formulário verifica-se o valor desse contador. Para conseguir isso vamos modificar o aplicativo de cálculo de números primos. No formulário principal incluímos na seção private um campo privado, FThreadsEmExecucao, que irá manter a quantidade de threads em execução. Também incluímos um método que irá responder ao evento OnTerminate. O método é o TratarEventoTerminate, que simplesmente irá decrementar em 1 o valor de FThreadEmExecucao. Posteriormente, no evento OnCreate do formulário setamos o valor de FThreadsEmExecucao para 0 (zero) e implementamos o método CloseQuery. Esse evento é disparado com o formulário for fechando, dando oportunidade para bloquear seu fechamento. **A Listagem 6** mostra a implementação de FormCloseQuery.

Listagem 5. Verificação de término no método EhPrimo

```
1. function CalculoPrimosThread.EhPrimo: boolean;
2. var
3.  iter: integer;
4. begin
5.  result := true;
6.  if FNumero < 0 then
7.  begin
8.   result := false;
9.   exit;
10. end;
11. if FNumero <= 2 then
12.  exit;
13. iter := 2;
14. while (iter < FNumero) and (not Terminated) do
15. begin
16.  if (FNumero mod iter) = 0 then
17.  begin
18.   result := false;
19.  end;
20.  iter := iter + 1;
21. end;
22. end;
```

Listagem 6. Implementação de FormCloseQuery

```
1. procedure TForm1.FormCloseQuery(Sender: TObject;
   var CanClose: Boolean);
2. begin
3.  CanClose := true;
4.  if FThreadsEmExecucao > 0 then
5.  begin
6.   if Application.MessageBox('Existem threads em execução.
          Deseja mesmo sair?',
7.          'Atenção',MB_YESNO) = ID_NO then
8.    CanClose := false;
9.
10. end;
11.end;
```

A linha 4 mostra a verificação realizada na contagem das threads, onde, se for maior que 0 (zero), um diálogo irá abrir solicitando uma confirmação. O campo FThreadsEmExecucao é acrescido sempre que uma nova thread é criada, como mostra a **Listagem 7**.

A principal diferença está nas linhas 9 e 10. Ao iniciar, o campo FThreadEmExecucacao tem seu valor acrescido de 1 e o evento OnTerminate recebeu sua implementação a partir do método TratarEventoTerminate. Ao executar a aplicação vemos que tudo realmente funciona.

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Varredura de strings

Com um exemplo mais complexo é possível demonstrar como utilizar várias threads para um objetivo comum, ao invés de apenas executar a mesma thread várias vezes. Nesse exemplo vamos varrer uma string e separá-la em várias partes, onde cada parte será um caractere da string, armazenado em um objeto TStringList. O objetivo é termos por exemplo a palavra 'clubedelphi' sendo analisada por uma thread "pai" que se encarregará de dividir a string em partes e cada parte será processada por threads "filhas". O resultado esperado pela thread pai é um TStringList contendo os caracteres 'c','l','u','b','e','d','e','l','p','h','i'. Assim, a complexidade pode estar em como fazer com que as threads filhas executem de tal forma que a thread pai consiga obter seus resultados na mesma ordem em que foram digitados.

Para começar, criamos então no novo aplicativo do tipo VCL Forms Application e organizamos sua interface de acordo com o que mostra a **Figura 3**.

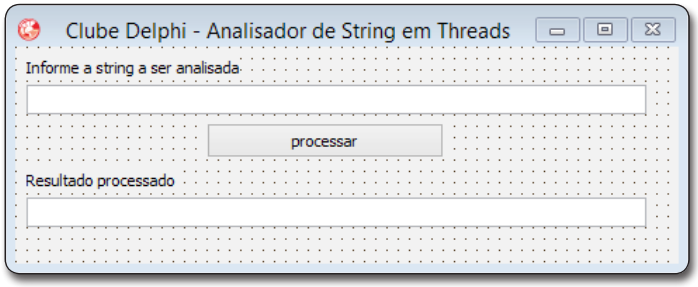

Figura 3. Formulário principal

Listagem 7. Criação das threads

```
1. procedure TForm1.btnCalcularClick(Sender: TObject);
2. var
3.   NovaThread: CalculoPrimosThread;
4. begin
5.   NovaThread := CalculoPrimosThread.Create(True);
6.   NovaThread.FreeOnTerminate := True;
7.   try
8.     NovaThread.Numero := StrToInt(editNumero.Text);
9.     FThreadsEmExecucao := FThreadsEmExecucao + 1;
10.    NovaThread.OnTerminate := TratarEventoTerminate;
11.    NovaThread.Resume;
12.  except on EConvertError do
13.    begin
14.      NovaThread.Free;
15.      ShowMessage('Não é um número válido!');
16.    end;
17.  end;
18. end;
```

Feito isso, adicionamos uma nova unit através do menu File>New>Other>Thread Object. O Delphi nos solicita o nome da classe, informamos o nome TAnalisadorString e ao salvar a unit, salvamos como AnalisadoresU. O assistente nesse momento criou a classe TAnalisadorString na sua forma mais básica. Agora adicionamos nessa unit a classe filha antes da declaração da classe TAnalisadorString, como mostra a **Listagem 8.**

O construtor da classe recebe a string a ser processada, essa string será passada pela thread TAnalisadorString. O campo Tag será utilizado para identificar o objeto dentro da lista de threads

filhas que a thread pai irá conter. O campo StringDeEntrada é usado no construtor para armazenar o parâmetro passado, já o campo Resultados é instanciado também no construtor e liberado da memória pelo destrutor Destroy. A **Listagem 9** mostra a implementação da thread filha.

Listagem 8. Classe TAnalisadorStringFilha

```
1. TAnalisadorStringFilha = class(TThread)
2. private
3.   StringDeEntrada : string;
4.   Resultados : TStringList;
5.   Tag : integer;
6. protected
7.    procedure Execute; override;
8. public
9.   constructor Create(const stringParcial : string); overload;
10.  destructor Destroy(); override;
11.end;
```

Listagem 9. Implementação da thread TAnalisadorStringFilha

```
1.  constructor TAnalisadorStringFilha.Create(const stringParcial: string);
2.  begin
3.    inherited Create(true);
4.    FreeOnTerminate := true;
5.    StringDeEntrada := stringParcial;
6.    Resultados := TStringList.Create;
7.  end;
8.
9.  destructor TAnalisadorStringFilha.Destroy;
10. begin
11.   FreeAndNil(Resultados);
12.   inherited;
13. end;
14.
15. procedure TAnalisadorStringFilha.Execute;
16. var
17.  c : char;
18. begin
19.  inherited;
20.  for c in StringDeEntrada do
21.  begin
22.    Sleep(Random(100));
23.    Resultados.Add(c);
24.  end;
25. end;
```

O construtor (linha 1) simplesmente inicializa os campos e seta a propriedade FreeOnTerminate para true, indicando que os recursos alocados pela thread serão liberados ao terminar sua execução. Já o método Execute (linha 15) está encarregado de quebrar a string em partes. Foi utilizado o comando for..in mas nada impediria o uso de outro comando, como o for..do. A cada caractere acessado da string é inserido no TStringList Resultados. Agora é o momento de implementar a thread pai. A **Listagem 10** mostra a definição da estrutura da classe.

Essa classe representa a classe pai, a thread que irá disparar as threads filhas. Por isso na linha 4 temos um array de TAnalisadorStringFilha e na linha 5 um outro array de TStringList que armazenará o resultado de cada thread. A classe pai também possui uma implementação (linha 7) que será associada ao evento Terminate de cada thread filha. Para ficar mais claro, a **Listagem 11** mostra a implementação de TAnalisadorString.

O construtor na linha 3 simplesmente executa o construtor base e configura a thread para limpar seus recursos alocados ao terminar sua execução, além de já inicializar os campos StringDeEntrada e Resultados.

Listagem 10. Classe TAnalisadorString

```
1.  TAnalisadorString = class(TThread)
2.  private
3.    StringDeEntrada: string;
4.    Filhas : array of TAnalisadorStringFilha;
5.    ResultadosFilhas : array of TStringList;
6.
7.    procedure FilhaOnTerminate(sender : TObject);
8.  protected
9.    procedure Execute; override;
10.   procedure DoTerminate; override;
11. public
12.   Resultados : TStringList;
13.   destructor Destroy(); override;
14.   constructor Create(const stringCompleta : string);
          overload;
15. end;
```

O método destrutor (linha 9) realiza a liberação da memória dos objetos instanciados. Por exemplo, a lista ResultadosFilhas é percorrida liberando cada elemento da memória. Quando a thread do AnalisadorString for abortada, ela deve abortar também as threads filhas. Isso é feito no método DoTerminate, onde na linha 24 o array Filhas é percorrido verificando quais thread ainda estão em execução, para então disparar seu método Terminate, forçando seu encerramento. O método Execute (linha 31) é responsável por criar duas threads filhas, essa quantidade foi definida na linha 33, na constante numFilhas. O texto a ser processado deve então ser dividido entre as duas threads, na linha 43 é calculado quantos caracteres serão destinados a cada uma. Na sequência, os arrays que armazenam as thread filhas e os resultados de cada thread, têm seu tamanho dimensionado para a quantidade de threads criadas. Na linha 46 são instanciados os objetos TStringList que conterão os resultados. O comando for executado posteriormente (linha 49) cria as threads filhas, já distribuindo as partes da string entre elas. Cada thread criada é armazenada em uma posição do array Filhas. Feito isso, o próximo comando for (linha 60) percorre o array Filhas, setando a propriedade Tag de cada thread com sua respectiva posição dentro do array e referencia o evento OnTerminate de cada uma com o método FilhaOnTerminate, para então executá-la. O loop while da linha 69 faz com que a thread pai aguarde as threads filhas terminarem seu processamento. Quando isso acontece, a lista de filhas é novamente percorrida para unir os resultados obtidos. Esses resultados são adicionados no TStringList Resultados. O método FilhaOnTerminate (linha 82) é disparado quando a thread encerra seu processamento, ou seja, quando a mesma já produziu um resultado. Nesse caso o resultado é transferido para o array ResultadosFilhas, para a posição de sua respectiva thread. Com toda a estrutura criada fica simples a execução das threads.

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Listagem 11. Implementação de TAnalisadorString

```
1.  constructor TAnalisadorString.Create(const stringCompleta: string);
2.  begin
3.    inherited Create(true);
4.    FreeOnTerminate := true;
5.    StringDeEntrada := stringCompleta;
6.    Resultados := TStringList.Create;
7.  end;
8.
9.  destructor TAnalisadorString.Destroy;
10. var
11.   i : integer;
12. begin
13.   for i := Low(Filhas) to High(Filhas) do ResultadosFilhas[i].Free;
14.   Filhas := nil;
15.   ResultadosFilhas := nil;
16.   FreeAndNil(Resultados);
17.   inherited;
18. end;
19.
20. procedure TAnalisadorString.DoTerminate;
21. var
22.   i : integer;
23. begin
24.   for i := Low(Filhas) to High(Filhas) do
25.     if NOT (Filhas[i].Terminated OR Filhas[i].Finished) then
26.       Filhas[i].Terminate;
27.
28.   inherited;
29. end;
30.
31. procedure TAnalisadorString.Execute;
32. const
33.   numFilhas = 2;
34. var
35.   i,
36.   charsPorFilha,
37.   inicio : integer;
38.   finalizado : boolean;
39.   parcial : string;
40. begin
41.   inherited;
42.
43.   charsPorFilha := Length(StringDeEntrada) DIV numFilhas;
44.   SetLength(Filhas, numFilhas);
45.   SetLength(ResultadosFilhas, numFilhas);
46.   for i := 0 to -1 + numFilhas do ResultadosFilhas[i] := TStringList.Create;
47.
48.   //cria e distribui as partes para as filhas
49.   for i := 0 to -1 + numFilhas do
50.   begin
51.     inicio := i * charsPorFilha + 1;
52.     if i = -1 + numFilhas then charsPorFilha := MaxInt;
53.
54.     parcial := Copy(StringDeEntrada, inicio, charsPorFilha);
55.
56.     Filhas[i] := TAnalisadorStringFilha.Create(parcial);
57.   end;
58.
59.   //executa as Filhas
60.   for i := Low(Filhas) to High(Filhas) do
61.   begin
62.     Filhas[i].OnTerminate := FilhaOnTerminate;
63.     Filhas[i].Tag := i;
64.     Filhas[i].Start;
65.   end;
66.
67.   //espera até todas Filhas terminarem
68.   finalizado := false;
69.   while NOT (Terminated OR finalizado) DO
70.   begin
71.     finalizado := true;
72.     for i := Low(Filhas) to High(Filhas) do
73.       finalizado := finalizado AND Filhas[i].Finished;
74.   end;
75.
76.   for i := Low(Filhas) to High(Filhas) do
77.   begin
78.     Resultados.AddStrings(ResultadosFilhas[i]);
79.   end;
80. end;
81.
82. procedure TAnalisadorString.FilhaOnTerminate(sender: TObject);
83. begin
84.   ResultadosFilhas[TAnalisadorStringFilha(Sender).Tag]
85.     .AddStrings(TAnalisadorStringFilha(Sender).Resultados);
86. end;
```

Listagem 12. Método AnalisadorTerminated

```
1. procedure TForm1.AnalisadorTerminated(sender: TObject);
2. begin
3.   TAnalisadorString(Sender).Resultados.Delimiter := ';';
4.   Edit2.Text :=
       TAnalisadorString(Sender).Resultados.DelimitedText;
5. end;
```

No formulário principal adicionamos uma referência à unit AnalisadoresU, também criamos na seção private um campo chamado analisador, do tipo TAnalisadorString, que foi criado anteriormente. A ideia geral de funcionamento é executar a thread TAnalisadorString e quando essa finalizar seu processamento, exibir o resultado no componente Edit2. Para processar o resultado criamos um procedure (AnalisadorTerminated) na seção private que será ligado ao evento OnTerminate da Thread. **A Listagem 12** mostra sua implementação.

Quando a thread disparar o evento, esse método é executado e em seu parâmetro sender é passada a instância da thread que disparou o evento. Assim, na linha 3 é realizado um type cast para o tipo TAnalisadoString que nos permite acessar a propriedade Resultados para então manipular o TStringList.

A execução do aplicativo então se resume agora ao clique do botão do formulário principal, mostrado na **Listagem 13**.

Listagem 13. Evento OnClick do botão do formulário principal

```
1. procedure TForm1.Button1Click(Sender: TObject);
2. begin
3.   analisador := TAnalisadorString.Create(Edit1.Text);
4.   analisador.OnTerminate := AnalisadorTerminated;
5.   analisador.Start;
6. end;
```

Na linha 3 o campo privado definido anteriormente é instanciado, recebendo como parâmetro o texto inserido no controle Edit1. Na sequência o método AnalisadorTerminated é então associado ao evento OnTerminate e a thread é iniciada em seu método Start. Ao executar o aplicativo é possível visualizar na

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

janela Thread Status do IDE que o aplicativo apresenta duas Threads (**Figura 4**).

Contudo, ao preenchermos o controle Edit1 e clicarmos no botão processar novas threads são criadas, como mostra a **Figura 5**.

Nota

Caso a janela Thread Status não esteja visível, acesse o menu View>Debug Windows>Threads.

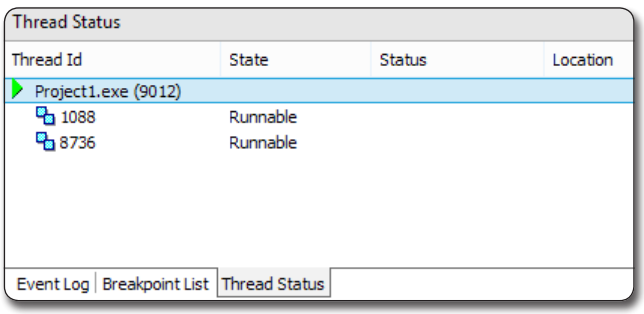

Figura 4. Threads em execução

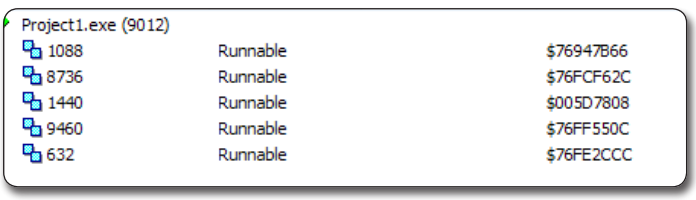

Figura 5. Novas threads criadas

É possível ver claramente que as threads que já existiam continuaram em execução e três novas foram criadas, essas são justamente a que o aplicativo criou. Uma ao instanciar e executar TAnalisadorString e as outras são as duas criadas pelo próprio TAnalisadorString. O resultado é visto na **Figura 6**.

Threads e banco de dados

Até o momento utilizamos threads apenas para realização de um cálculo e processamento de strings para ilustrar algumas tratativas e possibilidades, contudo, sabemos que o dia a dia de um desenvolvimento envolve conexão com banco de dados, execução de sentenças SQL e muito mais.

É muito comum utilizarmos apenas uma única thread onde uma sentença SQL é executada e esperamos seu término para recuperar seu resultado, para então executar outra consulta e assim por diante. Para acelerar o processamento, poderíamos ter threads independentes sendo executadas, todas recuperando informações de um banco de dados.

Para este exemplo criamos uma nova aplicação (File>New>VCL Forms Application) que se conectará a um banco de dados Firebird 1.5 utilizando os componentes IBX. O banco a ser utilizado é o employee.fdb, que acompanha as distribuições do Firebird. O aplicativo então permitirá escolher em componentes TComboBox departamentos e ao clicar em um botão, o aplicativo irá listar em componentes TListBox o primeiro nome dos funcionários dos departamentos escolhidos, mas como existem poucas informações, isso será feito em um loop de 100 iterações. Essa operação será dividida em dois grupos, sem a execução de threads e com a execução de threads e ao final do loop mencionado é disparado um sleep de um segundo para que seja comparável a execução com thread e sem thread. A **Figura 7** mostra o formulário principal.

No próprio formulário foi adicionado um componente TIBDatabase e TIBTransaction, e a conexão com o banco de dados foi realizada (**Figura 8**). No evento OnShow do formulário os componentes TComboBox (ComboDepto1, ComboDepto2, ComboThread1, ComboThread2) são preenchidos com as informações de código e nome de departamentos, como mostra a **Listagem 14**.

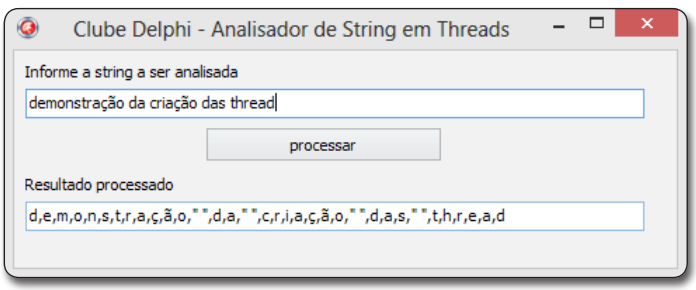

Figura 6. Resultado final da execução das threads

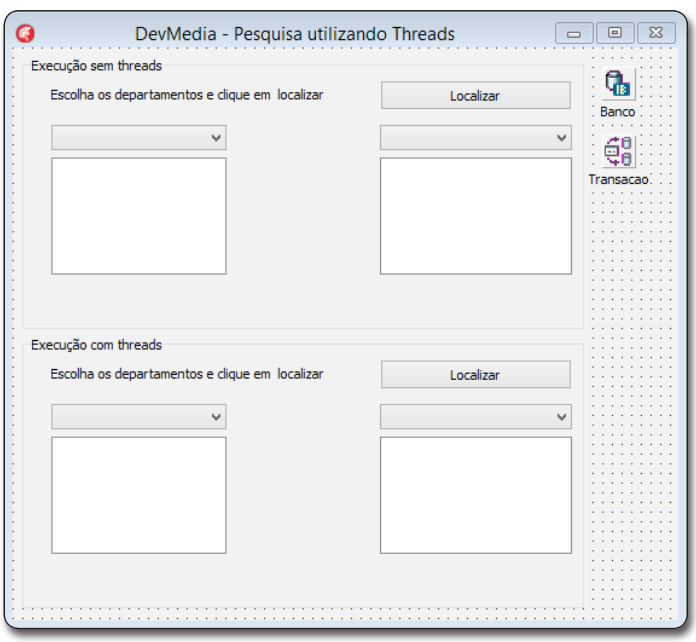

Figura 7. Formulário principal

Na linha 6 instanciamos um objeto do tipo TIBQuery, configurando-o na sequência para recuperar o campo dept_no e department da tabela department. Ao abrir a query a mesma é percorrida, em cada TComboBox é adicionado um item contendo o número e departamento (linhas 12 a 25) e ao término a propriedade ItemIndex de cada um deles é posicionado no primeiro elemento.

Nota

Adicionar na seção uses a unit IBX.IBQuery.

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

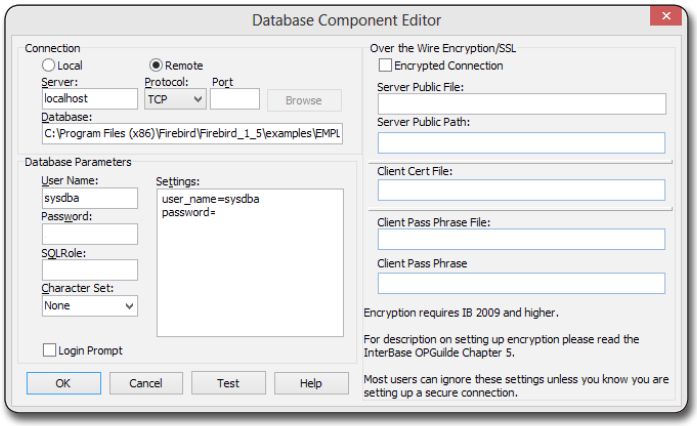

Figura 8. Conexão com o banco de dados (insira sua senha de conexão)

Listagem 14. Evento OnShow do formulário principal

```
1. procedure TForm1.FormShow(Sender: TObject);
2. var
3.   qry: TIBQuery;
4. begin
5.   banco.Open;
6.   qry := TIBQuery.Create(nil);
7.   qry.Database := banco;
8.   qry.Transaction := Transacao;
9.   qry.SQL.Add('select dept_no, department from department');
10.  qry.Open;
11.
12.  while not qry.Eof do
13.  begin
14.    ComboDepto1.Items.Add(qry.FieldByName('dept_no') .AsString + ' - ' +
15.      qry.FieldByName('department').AsString);
16.    ComboDepto2.Items.Add(qry.FieldByName('dept_no')
         .AsString + ' - ' +
17.      qry.FieldByName('department').AsString);
18.    ComboThread1.Items.Add(qry.FieldByName('dept_no') .AsString + ' - ' +
19.      qry.FieldByName('department').AsString);
21.    ComboThread2.Items.Add(qry.FieldByName('dept_no') .AsString + ' - ' +
22.      qry.FieldByName('department').AsString);
23.
24.    qry.Next;
25.  end;
26.  ComboDepto1.ItemIndex := 0;
27.  ComboDepto2.ItemIndex := 0;
28.  ComboThread1.ItemIndex := 0;
29.  ComboThread2.ItemIndex := 0;
30.
31.  qry.Close;
32.  FreeAndNil(qry);
33. end;
```

Nesse momento vamos executar a consulta sem threads. No evento OnClick do botão Localizar do TGroupBox "Execução sem threads" adicionamos o código da **Listagem 15**.

Na linha 9 a variável qry é instanciada e seu SQL é adicionado. Após abrir a consulta, o comando for (linha 18) é executado e nele o resultado da consulta é percorrido 100 vezes. Ao finalizar, a consulta é fechada (linha 27), o comando Application.ProcessMessages é executado para que o Windows possa processar as mensagens de atualização de tela e na linha 30 o sleep é disparado, efetuando uma pausa de um segundo para que fique perceptível que quando executamos sem as threads, o processamento é linear. Então o processo todo se repete para a segunda query, com exceção de uma segunda chamada ao sleep que não se faz necessário. Ao executar a aplicação nesse momento, e escolher os departamentos e clicar em localizar, é possível ver que o programa primeiramente preenche um controle TListBox para então preencher o segundo, de forma sequencial. Vamos agora criar a thread que nos permitirá fazer o mesmo, mas de forma concorrente.

A primeira coisa a se fazer é criar uma classe que herde de TThread, assim, no próprio formulário principal antes da declaração da classe TForm1, declaramos a classe TConsultaFuncionariosThread como vista na **Listagem 16**.

A classe possui dois campos privados (linhas 3 e 4). FListBox armazenará o controle TListBox do formulário e FCodigoDepartamento o código do departamento que será passado via construtor, como mostra **Listagem 17**.

O construtor foi modificado para receber o código de departamento escolhido na tela e o componente TListBox existente que armazenará os resultados. Em sua implementação é executado o construtor base (linha 4) onde é passado o parâmetro True, indicando que a thread é criada em suspensão, ou seja, é criada mas não é executada de imediato. Na sequência a propriedade FreeOnTerminate recebe True, indicando que os recursos alocados devem ser liberados, a prioridade (**BOX 1**) da thread é ajustada para tpNormal, ou seja, uma prioridade normal e os dois campos privados tem seu valor setado.

BOX 1. TThreadPriority

A propriedade Priority da classe TThread é do tipo TThreadPriority, que representa os níveis de prioridade no escalonamento. Existem os seguintes níveis: tpIdle, tpLower, tpNormal, tpHigher, tpHighest, tpTimeCritical. Respectivamente esses valores representam: executar uma thread somente quando o Windows estiver no estado idle, executar uma thread dois pontos abaixo do normal, um ponto abaixo do normal, prioridade normal, um ponto acima do normal, dois pontos acima do normal e, prioridade máxima acima de outras threads.

O método Execute é responsável por realizar as consultas, sua implementação pode ser vista na **Listagem 18**.

Nas linhas 3 e 4 são declaradas variáveis que irão implementar uma conexão com o banco de dados, ou seja, cada thread a ser executada irá criar uma nova conexão. Isso é feito para garantir o que chamados de Thread-Safety (**BOX 2**). A criação da conexão é feita entre as linhas 11 e 26. Na sequência, de forma similar ao código sem thread, a query é instanciada e tem seu parâmetro passado (linhas 30 à 36).

BOX 2. Thread-Safety no Firebird

No Firebird o maior nível de concorrência suportado está no nível de conexão, isto é, é seguro utilizar uma conexão em somente uma thread por vez, embora uma mesma conexão possa ser manipulada por diferentes threads de forma serial (uma por vez) e diferentes conexões possam ser manipuladas por diferentes threads em paralelo. Por exemplo, em um aplicativo multithread é seguro para uma thread criar e utilizar uma conexão e depois liberá-la e depois uma outra thread reutilizar essa conexão mas, não é seguro que ambas as threads utilizando uma mesma conexão ao mesmo tempo.

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Após a consulta ser realizada, o resultado é processado em um laço for (linha 38) que executará por 100 vezes o preenchimento do controle TListBox passado no construtor. Feito isso a consulta é fechada, o comando Application.ProcessMessages

Listagem 15. Recuperando informações sem thread

```
1.  procedure TForm1.Button1Click(Sender: TObject);
2.  var
3.    qry: TIBQuery;
4.
5.    x: integer;
5.  begin
6.    ListBox1.Clear;
7.    ListBox2.Clear;
8.
9.    qry := TIBQuery.Create(nil);
10.   qry.Database := banco;
11.   qry.Transaction := Transacao;
12.   qry.SQL.Add('select first_name from employee');
15.   qry.SQL.Add('where dept_no = :departamento');
14.
15.   qry.ParamByName('departamento').AsString :=
16.           copy(ComboDepto1.Text,1,3);
17.   qry.Open;
18.   for x := 1 to 100 do
19.   begin
20.     while not qry.Eof do
21.     begin
22.       ListBox1.Items.Add(
            qry.FieldByName('first_name').AsString);
23.       qry.Next;
24.     end;
25.     qry.First;
26.   end;
27.   qry.Close;
28.
29.   Application.ProcessMessages;
30.   sleep(1000);
31.
32.   qry.ParamByName('departamento').AsString :=
33.         copy(ComboDepto2.Text,1,3);
34.   qry.Open;
35.   for x := 1 to 100 do
36.   begin
37.     while not qry.Eof do
38.     begin
39.       ListBox2.Items.Add(
          qry.FieldByName('first_name').AsString);
40.       qry.Next;
41.     end;
42.     qry.First;
43.   end;
44.   qry.Close;
45.   FreeAndNil(qry);
46. end;
```

Listagem 16. Thread de consulta

```
1.  TConsultaFuncionariosThread = class(TThread)
2.  private
3.    FListBox : TListBox;
4.    FCodigoDepartamento: string;
5.  protected
6.    procedure Execute; override;
7.  public
8.    constructor Create(CodigoDepartamento: string;
        ListBox: TListBox);reintroduce;
9.  end;
```

Listagem 17. Implementação do construtor

```
1. constructor TConsultaFuncionariosThread.Create(CodigoDepartamento: string;
2. ListBox: TListBox);
3. begin
4.   inherited Create(true);
5.   FreeOnTerminate := true;
6.   Priority := tpNormal;
7.   FCodigoDepartamento := CodigoDepartamento;
8.   FListBox := ListBox;
9. end;
```

Listagem 18. Implementação do método Execute

```
1.  procedure TConsultaFuncionariosThread.Execute;
2.  var
3.    fbBanco: TIBDataBase;
4.    fbTrans: TIBTransaction;
5.    qry: TIBQuery;
6.
7.    x: integer;
8.  begin
9.    inherited;
10.
11.   fbBanco := TIBDatabase.Create(nil);
12.   fbBanco.DatabaseName := 'localhost:C:\Program Files (x86)\Firebird\
        Firebird_1_5\examples\EMPLOYEE.FDB';
13.   fbBanco.LoginPrompt := false;
14.   fbBanco.ServerType := 'IBServer';
15.   fbBanco.Params.Add('user_name=sysdba');
16.   fbBanco.Params.Add('password=cont10rol');
17.
18.   fbTrans := TIBTransaction.Create(nil);
19.   fbTrans.Params.Add('read_committed');
20.   fbTrans.Params.Add('rec_version');
21.   fbTrans.Params.Add('nowait');
22.
23.   fbTrans.DefaultDatabase := fbBanco;
24.   fbBanco.DefaultTransaction := fbTrans;
25.   fbBanco.Open;
26.   fbTrans.Active := true;
27.
28.   FListBox.Clear;
29.
30.   qry := TIBQuery.Create(nil);
31.   qry.Database := fbBanco;
32.   qry.Transaction := fbTrans;
33.   qry.SQL.Add('select first_name from employee');
34.   qry.SQL.Add('where dept_no = :departamento');
35.
36.   qry.ParamByName('departamento').AsString :=  FCodigoDepartamento;
37.   qry.Open;
38.   for x := 1 to 100 do
39.   begin
40.     while not qry.Eof and not Terminated do
41.     begin
42.       FListBox.Items.Add(
            qry.FieldByName('first_name').AsString);
43.       qry.Next;
44.     end;
45.     qry.First;
46.   end;
47.   qry.Close;
48.   FreeAndNil(qry);
49.   Application.ProcessMessages;
50.   sleep(1000);
51.
52.   fbTrans.Active := false;
53.   fbBanco.Close;
54.   FreeAndNil(fbTrans);
55.   FreeAndNil(fbBanco);
56. end;
```


DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

é executado para forçar a aplicação a ler as mensagens pendentes do sistema operacional e responde-las e um sleep de 1 segundo é aplicado. Por fim, a conexão é desfeita e os objetos são liberados da memória.

Para colocar tudo isso em prática é necessário implementar o evento onClick do botão Localizar do TGroupBox "Executar com Threads", como mostra a **Listagem 19**.

Listagem 19. Utilizando as threads

```
1. procedure TForm1.Button2Click(Sender: TObject);
2. var
3.  thread1, thread2: TConsultaFuncionariosThread;
4. begin
5.  thread1 := TConsultaFuncionariosThread.Create(
6.            copy(ComboThread1.Text,1,3),
7.            ListBox3);
8.
9.  thread2 := TConsultaFuncionariosThread.Create(
10.           copy(ComboThread2.Text,1,3),
11.           ListBox4);
12.  thread1.Start;
13.  thread2.Start;
14. end;
```

Temos duas variáveis do tipo TConsultaFuncionariosThread, elas são devidamente instanciadas (linhas 5 e 9), são passados como parâmetro o código de departamento exibido em cada TComboBox destinado às threads e os controles TListBox que receberão os resultados. Nas linhas 12 e 13 esses objetos thread são iniciados.

Ao executar fica claro que as duas consultas são executadas de forma concorrente porque não acontece a pausa de 1 segundo criada (**Listagem 18**, linha 50). A **Figura 9** mostra uma comparação na execução das duas abordagens.

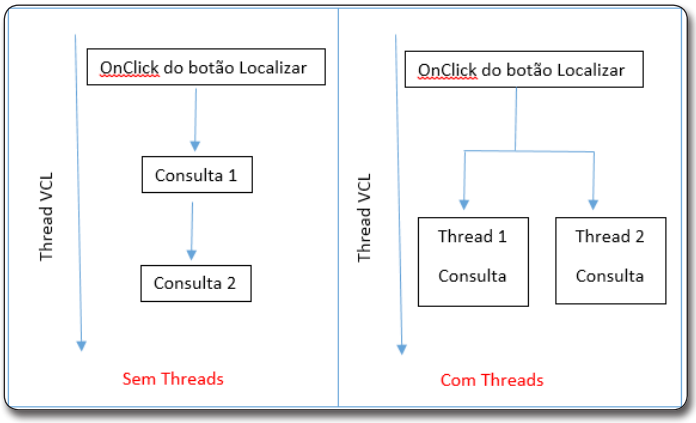

Figura 9. Visualização da execução

A forma de criação das threads utilizou a classe base TThread, que é compatível com várias versões do Delphi, por exemplo, antes da versão 7 até a versão XE 6 (utilizada nesse artigo). Contudo, a linguagem Delphi sofreu modificações, melhorias e com isso foram criadas bibliotecas para criar e gerenciar Threads de uma maneira mais moderna e eficiente.

Autor

Paulo Quicoli
pauloquicoli@gmail.com
Editor da revista ClubeDelphi. Formado em processamento de dados pela FATEC-TQ. Atua como analista programador de sistemas da Siplan Control-M (www.siplancontrolm.com.br) unidade Jaboticabal. Blog: http://quicoli.worpress.com

Você gostou deste artigo?

Dê seu voto em ***www.devmedia.com.br/clubedelphi/feedback***

Ajude-nos a manter a qualidade da revista!

Como incorporar arquivos de recursos em executáveis no Delphi

Adicione executáveis e arquivos à sua aplicação

Durante o desenvolvimento de um software podemos fazer uso de DLLs, criar modelos de documentos e tudo isso precisa ser instalado junto com nosso software. Para facilitar uma implantação ou atualização, podemos incluir esses arquivos, ou como são chamados, recursos, dentro de nosso executável e no momento de sua execução, extraí-los.

Vídeo, som, imagem ou qualquer arquivo binário, por exemplo, pode ser facilmente incorporado a um executável feito em Delphi através do uso da diretiva de recursos (RES - Resource). É muito importante olhar com cuidado o uso dessa técnica, porque esses arquivos de recurso podem até dobrar o tamanho do executável. Também podemos fazer uso de outras possibilidades, como uso de pacotes BPL ou bibliotecas DLL dependendo da necessidade do projeto.

Estes recursos são parte do arquivo executável e são carregados ao mesmo tempo em que a aplicação é executada. O seu carregamento e descarregamento são realizados pela memória livre. Então não há um limite de números de recursos a serem carregados ou descarregados.

Fique por Dentro

Este artigo é útil em diversos tipos de aplicações que necessitam periodicamente de mudanças, seja para correção de erros, melhorias, ou adequações. Para estes casos, utiliza-se alguns recursos, entre eles o de distribuir com a aplicação outro executável. Isto permite verificar se há mudanças a realizar, validar a versão do software entre outras. Essas aplicações e diversas outras, utilizam essa técnica de incluir outros arquivos, músicas, imagens, cursores personalizados, arquivos de texto e outros executáveis "embutidos" no executável principal. Em Delphi podemos fazer isso utilizando recursos.

Arquivos de script de recurso (.rc)

Um arquivo de script de recursos nada mais é que um arquivo de texto simples com a extensão .rc. Este arquivo é compilado por uma ferramenta, para então gerar o arquivo res. Para isso, geralmente utilizamos a Borland Resource Compiler com a finalidade de criá-lo (res). A estrutura de um arquivo rc para incorporar arquivos é composta basicamente por três características, sendo elas:

• **Nome do recurso -** Uma identificação para o recurso a ser usado pelo executável. Para criar esse nome, pode-se adotar qualquer caractere, inclusive números de 0 a 9. Só não são permitidos caracteres especiais (^, %, #, @, !, ~, *, &) e nem qualquer tipo de acentuação.

• **Tipo do recurso -** Para identificar o tipo do recurso, ou seja, o tipo do arquivo que está sendo incorporado. Já existem alguns tipos de recursos predefinidos como o Bitmap, Icon, Cursor e etc. Para outros tipos de arquivos como executáveis, pode-se definir o tipo File.

• **Caminho para o recurso** – Local onde se encontra o arquivo que se deseja utilizar como recurso. O diretório deve estar entre os caracteres de aspa dupla, ou será emitido um erro na hora de compilar o arquivo.

Internacionalizando a aplicação com o script de recurso

Além do que já foi citado até aqui, é possível também internacionalizar a aplicação com um arquivo de recurso. Para isso deve-se declarar no arquivo de recurso uma tabela de Strings (String Table). Nessa tabela de Strings inserimos todos os textos para fazer a internacionalização. Existem outras maneiras de se internacionalizar um projeto desenvolvido em Delphi. Além deste, os mais comuns são: o uso de DLLs, ResourceString e usando banco de dados para armazenamento dos textos. No entanto, se o projeto não for muito grande e não conter muitas telas e objetos, torna-se vantajoso fazer o uso do Resource para essa internacionalização, visto que um arquivo de Resource, com poucas Strings, não pesaria em nada a aplicação, sendo o tamanho insignificante. Uma forma simples de fazer uma tabela de Strings

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

para esse propósito seria colocar os textos dos objetos com uma quebra de linha entre as línguas.

A estrutura de uma String Table é relativamente simples, composta por um identificador e a String em si. Veja a seguir na **Listagem 1** um exemplo de sua estrutura.

Observe que utilizamos a palavra STRINGTABLE para representar a tabela de strings, seu início e fim são delimitados pelas chaves e cada palavra a ser internacionalizada recebe um número de identificação. Veja que foi inserida uma quebra (linha em branco) entre os idiomas português e inglês para facilitar a visualização. O identificador numérico é o que será usado para recuperarmos essa String. Com uma chamada à função LoadString (**BOX 1**) passamos esse identificador para que a string seja retornada.

Compilando o arquivo de script de recurso

Para fazer uso das funcionalidades que foram definidas no arquivo de script de recurso da **Listagem 1** temos primeiramente que compilá-lo como já mencionado, para que então seja gerado o arquivo RES.

Usaremos o compilador de recursos da Borland (brcc32.exe). O mesmo se encontra no diretório bin do Delphi. Ao compilar, o arquivo de script de recurso será gerado no mesmo local onde se encontra o arquivo RES com base nos conteúdos definidos no arquivo RC. O arquivo RES será gerado com o mesmo nome do arquivo RC. Para facilitar a criação desse arquivo, é mais fácil colocá-lo no diretório bin do Delphi. Como um projeto Delphi também gera um arquivo RES com o mesmo nome, temos que nomear este RC com um nome diferente do projeto para que quando compilarmos arquivo RC, seja gerado um RES com outro nome, caso contrário, teremos dois arquivos RES com o mesmo nome.

Para efetuar esta compilação, basta abrir um prompt de comando no diretório bin do Delphi e usar o brcc32 com o nome do arquivo RC. Supondo que o arquivo RC tenha o nome de ArtResource .rc, e já se encontre no diretório bin, bastaria digitar o comando brcc32 ArtResource.rc. Assim sendo, será gerado um arquivo com o mesmo nome, porém, com a extensão .RES. Com isso, teríamos o arquivo RES para fazermos uso em nossas aplicações. Um exemplo da compilação de um arquivo RC pode ser visto na **Figura 1.**

Listagem 1. String Table

```
STRINGTABLE
{
1000, "Português"
1001, "Sim"
1002, "Não"

2000, "Inglês"
2001, "Yes"
2002, "No"
}
```

BOX 1. LoadString

Essa função, que está contida na DLL User32.dll, carrega através do executável um recurso de texto que se encontra em um Resource a ele associado. Ao usarmos esta chamada à API, passamos o parâmetro Hinstance que é o identificador para uma instância do nosso recurso. Além disso, passa-se também o parâmetro ID, que é o identificador da string que queremos carregar. No caso do exemplo exposto anteriormente, seria um dos valores entre 1000 e 2002. Para finalizar, são passados mais dois parâmetros (LpBuffer e NBufferMax). LpBuffer é o responsável por receber a String. Essa String terá um caractere nulo para indicar o seu término. Já NbufferMax é o parâmetro da memória em caracteres. O retorno dessa função é a quantidade de caracteres copiados para LpBuffer, não incluindo o caractere nulo de terminação ou zero, caso não exista a String no recurso.

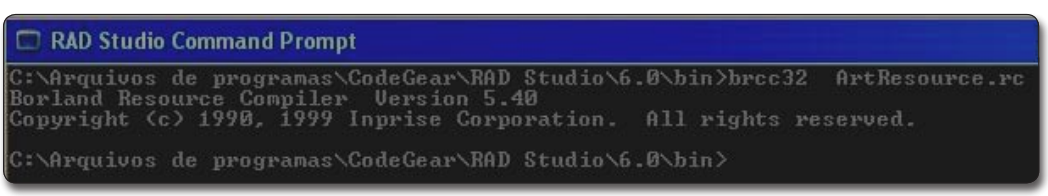

Figura 1. Compilação de um arquivo Rc

Nota

O compilador brcc32 talvez seja o mais utilizado por desenvolvedores Delphi, no entanto, há outros editores e compiladores de recursos como o Windres, Borland Resource WorkShop, Microsoft Windows Resource Compiler (RC), entre outros.

Usando o arquivo de resource

Para usar esses recursos que foram definidos no arquivo de script de recurso e que agora estão embutidos nesse arquivo RES basta adicionarmos à nossa aplicação Delphi uma diretiva do compilador no código-fonte do projeto Delphi. Para visualizar o código-fonte do .dpr usamos o menu View> Source do Delphi.

No código fonte adicione a diretiva logo abaixo da diretiva de compilação {$R *.res}. Utilizando como exemplo, um recurso chamado (ArtResource), ficaria da seguinte forma:

```
{$R *.res}
{$R ArtResource}
```

Ao compilar o projeto, esse resource é automaticamente incorporado à aplicação juntamente com todos os recursos contidos nele.

Criando um arquivo res para o exemplo

Para ficar um pouco mais claro o entendimento dessa diretiva e do arquivo de Script de recursos, iremos criar esse arquivo e compilá-lo com a finalidade de gerar esse resource e usá-lo em uma aplicação. Como já visto anteriormente, é simplesmente um arquivo de texto com a extensão .rc. Então se pode facilmente escrevê-lo em um bloco de notas, a exemplo do arquivo rc da **Listagem 2.**

Após ter escrito esse arquivo em bloco de notas, basta salvá-lo com a extensão rc. O nome dado ao exemplo foi ArtResource.rc.

O conteúdo definido no resource apresentado na **Listagem 2** é composto por três imagens: duas do tipo Bitmap (Img1, Img2) e uma do tipo Jpeg (Img3).

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Listagem 2. Arquivo texto (script) para criação do resource

```
IMG1 BITMAP "C:\Arquivos de programas\Arquivos comuns\CodeGear Shared\
Images\Splash\16Color\ATHENA.bmp"

IMG2 BITMAP "C:\Arquivos de programas\Arquivos comuns\CodeGear Shared\
Images\Splash\16Color\SKYLINE.bmp"

IMG3 JPEG "C:\Documents and Settings\Administrador\Desktop\Busca.jpg"

ICONE ICON "C:\Arquivos de programas\Arquivos comuns\CodeGear Shared\
Images\Icons\FACTORY.ico"

CURSR CURSOR"C:\Arquivos de programas\Arquivos comuns\CodeGear Shared\
Images\Cursors\HANDPNT.cur"

DOCTXT FILE "C:\Documents and Settings\Administrador\Desktop\Doctxt.txt"

MUSICA WAVE "C:\WINDOWS\Media\ringout.wav"

PROGRAMA FILE "C:\Documents and Settings\Administrador\Desktop\
PrgSecundario.exe"

STRINGTABLE
{
1000, "Português"
1001, "Imagem Nº1"
1002, "Imagem Nº2"
1003, "Icone"
1004, "Cursor"
1005, "Documento Txt"
1006, "Musica Wave"
1007, "Programa Secundário"
1008, "Programa Uso de Resource"
1009, "Imagem Nº3"
2000, "Inglês"
2001, "Picture Nº1"
2002, "Picture Nº2"
2003, "Icon"
2004, "Cursor"
2005, "Txt Document"
2006, "Music Wave"
2007, "Secondary Program"
2008, "Use Program Resource"
2009, "Picture Nº3"
3000, "Francês"
3001, "Photo Nº1"
3002, "Photo Nº2"
3003, "Icône"
3004, "Curseur"
3005, "Document Txt"
3006, "Musique Wave"
3007, "Programme Secondaire"
3008, "L'Utilisation des Ressources du Programme"
3009, "Photo Nº3"
}
```

Algo muito importante a ser lembrado é que o caminho desse arquivo no disco ou na rede deve estar entre aspas. Seguindo ainda a explanação do arquivo, definimos um Icone, um Cursor, um arquivo de texto simples Txt (DocTxt), um arquivo de música do tipo Wave (Musica) e, para finalizar, foi adicionado outro executável (PrgSecundario.exe).

Esses foram os arquivos inclusos no resource. Como já mencionado, conforme de arquivos adicionados como recurso dentro da aplicação, o tamanho cresce consideravelmente.

Após a definição dos arquivos foi criada uma tabela de Strings. Essa tabela será a responsável por traduzir os botões de um formulário juntamente ao seu título (Caption). Além disso, estamos definindo três línguas (Português, Inglês e Francês) e nove informações para cada língua. Essas informações, são referentes aos oito arquivos definidos (Img1, Img2, Img3, Icone, Cursr, DocTxt, Musica, Programa) e ao título da aplicação.

Após a explicação do arquivo de script de recurso, basta compilá-lo, para que seja gerado o arquivo ArtResource.Res. Esse executável adicionado ao arquivo res é um aplicativo em Delphi composto por uma tela simples que exibe um texto em um Memo, conforme mostra a **Figura 2**, no entanto, poderia ser qualquer outro aplicativo.

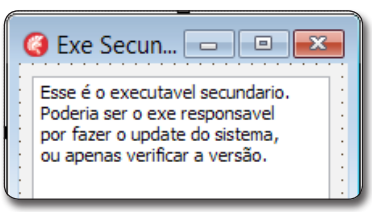

Figura 2. Executável incorporado ao aplicativo

Nota

Como o projeto é composto apenas por um aplicativo e um Memo. Não abordaremos sua criação, porém, o código deste aplicativo estará disponível juntamente do código-fonte do artigo.

Iniciaremos criando a aplicação que fará uso deste resource. Para isso, crie um novo projeto nomeando-o como PrgResource. Modifique o nome do formulário principal para Frm_Principal e sua unit para Unt_Principal. Após isso, criaremos uma nova unit chamada de Unt_Traduzir que será a responsável por traduzir os Captions dos componentes da nossa aplicação, como mostra a **Listagem 3.**

Nesse código podemos ver que a classe TTraduzir será muito simples, contendo apenas dois métodos (BuscaLinguas e BuscaTexto), nas linhas 07 e 08 respectivamente.

Listagem 3. Unt_Traduzir buscará as línguas e os textos do Resource

```
01 unit Unt_Traduzir;

02 interface

03 uses
04  Windows, Classes;

05 type
06  TTraduzir = class
07    class procedure BuscaLinguas(const Strings: TStrings);
08    class function BuscaTexto(const Idx, Position: Integer): String;
09  end;

10 implementation
```


DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

A seguir, nas **Listagens 4** e **5** temos a implementação desses dois métodos. Note que temos as chamadas de class procedure e class function, que são chamados métodos de classe (**BOX 2**).

BOX 2. Métodos de classe

Diferentemente das chamadas convencionais, aqui usamos métodos de classe. Os métodos de classe são comumente utilizados para criações de Units especificas de "funções" de auxílio em aplicações. A grande vantagem é que neste caso não precisamos instanciar a classe para utilizar seus métodos, bastando apenas utilizar a sintaxe: TClasse.NomeMétodo.

Listagem 4. Implementação do método BuscaLinguas

```
01 class procedure TTraduzir.BuscaLinguas(const Strings: TStrings);
02 const
03   Intervalo = 1000;
04 var
05   Buffer: array[0..255] of Char;
06   Indice, Position: Integer;
07 begin
08   Position := Intervalo;
09   Strings.Clear;
10   Indice := LoadString(HInstance, Position, Buffer, SizeOf(Buffer));
11   while (Indice <> 0) do
12   begin
13     Strings.AddObject(Buffer, TObject(Position));
14     Position := Position + Intervalo;
15     Indice := LoadString(HInstance, Position, Buffer, SizeOf(Buffer));
16   end;
17 end;
```

Nesse código temos a implementação do método de classe BuscaLinguas. Nele, carregaremos as linguagens disponíveis em um TRadioGroup. Estamos definindo um intervalo que é de 1000 (linhas 02 e 03) ou seja, a cada intervalo deste, teremos uma nova língua, e claro que os valores que estiverem dentro desses intervalos, serão os textos referentes à língua que usaremos para a tradução.

Sendo assim, para pegarmos essas línguas usamos a função LoadString (linha 10). Passando esse intervalo de 1000, é realizado um loop (linha 11) e enquanto houver esse intervalo (língua) no Resource, vamos obtendo o seu conteúdo e incrementando o intervalo para verificar se existe outra língua.

A seguir, na **Listagem 5** temos a implementação do segundo método definido na classe TTraduzir.

Listagem 5. Implementação do método BuscaTexto

```
01 class function TTraduzir.BuscaTexto(const Idx, Position: Integer):  String;
02 var
03   Buffer : array[0..255] of Char;
04   Indice : Integer;
05 begin
06   Result := '';
07   Indice := LoadString(HInstance, Idx + Position, Buffer, SizeOf(Buffer));
08   if (Indice <> 0) then
09     Result := Buffer;
10 end;
```

Nessa listagem temos a implementação do método BuscaTexto, responsável por buscar o nosso texto desejado. Passamos dois parâmetros na chamada do mesmo (Idx e Position – linha 01). O parâmetro Idx refere-se ao número de identificação da língua. Exemplo: (1000 = Português, 2000 = Inglês). O parâmetro Position é a identificação da posição de qual texto queremos. Exemplo (1 = IMG1, 2 = IMG2 e etc.). Ou seja, se quisermos o texto em Francês, que está na posição 6. Chamaremos essa função, passando os parâmetros (3000, 6). No caso do nosso resource feito anteriormente, o resultado seria (Musique Wave).

Voltando ao aplicativo principal

Em nosso formulário principal iremos adicionar os componentes necessários para a criação do exemplo. Sendo assim, adicionamos ao formulário oito Buttons, um Image, um RadioGroup e um Memo. O layout proposto para a aplicação é o da **Figura 3**.

Primeiramente, definiremos o método WinExecAndWait na sessão private do formulário. O cabeçalho do método é o seguinte:

```
function WinExecAndWait(const Path: PChar; const Visibility: Word; const
  Wait: Boolean) : Boolean;
```

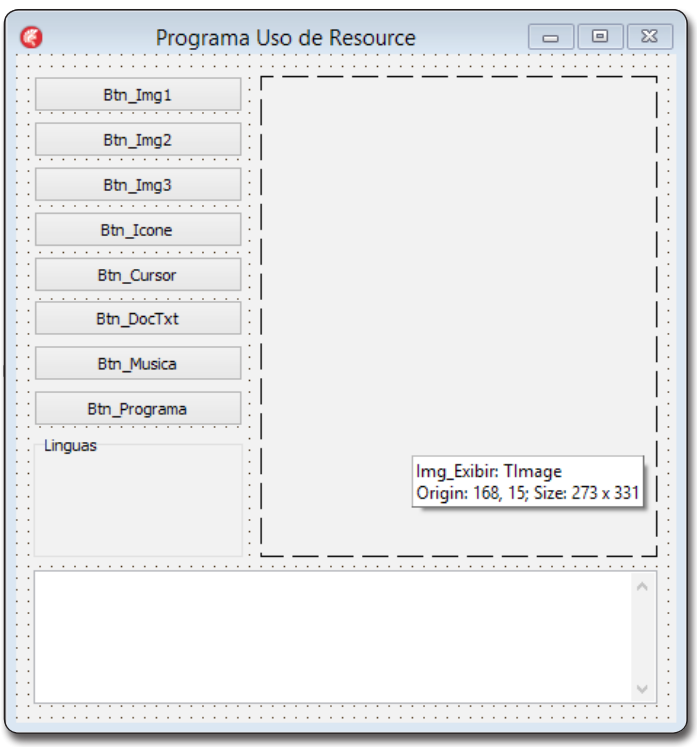

Figura 3. Tela do aplicativo

A implementação do método encontra-se na **Listagem 6**.

Entre as linhas 03 e 04 temos declaradas duas variáveis. A variável ProcessInformation é responsável por receber a estrutura de um processo que criaremos com a finalidade de executar esse programa secundário. Note que entre as linhas 07 e 17 que há as configurações referentes à variável StartupInfo, que trata-se da

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

estrutura da janela do aplicativo que passamos no momento da criação do processo. Na linha 18 o processo é criado usando o CreateProcess, que por sua vez também cria a thread principal e executa o aplicativo informado no Resource, que no nosso caso é o programa secundário. Para finalizarmos, é chamado o método CloseHandle para liberar esses processos (linhas 25 e 26).

Continuando a codificação da aplicação principal, faremos a implementação do evento OnCreate que simplesmente carrega as línguas disponíveis dentro do RadioGroup e, caso haja itens, automaticamente é setada a primeira posição do mesmo, conforme mostra a **Listagem 7**.

Listagem 6. Método WinExecAndWait

```
01 function TFrm_Principal.WinExecAndWait(const Path: PChar;
     const Visibility: Word; const Wait: Boolean): Boolean;
02 var
03   ProcessInformation: TProcessInformation;
04   StartupInfo: TStartupInfo;
05 begin
06   FillChar(StartupInfo, SizeOf(TStartupInfo), 0);
07   with StartupInfo do
08   begin
09     cb        := SizeOf(TStartupInfo);
10     lpReserved  := NIL;
11     lpDesktop   := NIL;
12     lpTitle     := NIL;
13     dwFlags     := STARTF_USESHOWWINDOW;
14     wShowWindow := Visibility;
15     cbReserved2 := 0;
16     lpReserved2 := NIL
17   end;
18   Result := CreateProcess(NIL, Path, NIL, NIL, FALSE,
                  NORMAL_PRIORITY_CLASS,
                  NIL, NIL, StartupInfo, ProcessInformation);
19   if Result then
20   begin
21     with ProcessInformation do
22     begin
23       if Wait then
24         WaitForSingleObject(hProcess, 100);
25       CloseHandle(hThread);
26       CloseHandle(hProcess);
27     end;
28   end;
29 end;
```

Listagem 7. Evento FormCreate

```
01 procedure TFrm_Principal.FormCreate(Sender: TObject);
02 begin
03   TTraduzir.BuscaLinguas(Rgp_Linguas.Items);
04   if (Rgp_Linguas.Items.Count >= 0) then
05     Rgp_Linguas.ItemIndex := 0;
06 end;
```

Seguindo a sequência dos botões definidos, começaremos pelos eventos OnClick dos botões Btn_Img1 e Btn_Img2 respectivamente. O código de ambos é mostrado na **Listagem 8**.

Observe que a implementação é a mesma dos dois botões, a única diferença é a tag 'IMG1' e 'IMG2', que refere-se ao Resource. O método LoadBitmap utiliza o Handle para o carregamento da imagem. Repare que o resultado é automaticamente atribuído ao objeto Img_Exibir em ambos os casos.

Para o botão 3 (Btn_Img3) faremos uma listagem a parte, afinal, diferentemente das situações anteriores, aqui carregamos um JPEG, sendo assim, implemente o evento OnClick do mesmo com o código da **Listagem 9**.

Listagem 8. Evento OnClick que carrega as imagens

```
01 procedure TFrm_Principal.Btn_Img1Click(Sender: TObject);
02 begin
03   Img_Exibir.Picture.Bitmap.Handle := LoadBitmap(HInstance, 'IMG1');
04 end;

05 procedure TFrm_Principal.Btn_Img2Click(Sender: TObject);
06 begin
07   Img_Exibir.Picture.Bitmap.Handle := LoadBitmap(HInstance, 'IMG2');
08 end;
```

Listagem 9. Evento OnClick do Btn_Img3 que carrega um JPEG

```
01 procedure TFrm_Principal.Btn_Img3Click(Sender: TObject);
02 var
03   Res: TResourceStream;
04   MemStream: TMemoryStream;
05   hFind, hRes: THandle;
06 begin
07   hFind := FindResource(HInstance, 'IMG3', 'JPEG');
08   if (hFind <> 0) then
09   begin
10     hRes := LoadResource(HInstance, hFind);
11     if (hRes <> 0) then
12     begin
13       MemStream := TMemoryStream.Create;
14       Res       := TResourceStream.Create(HInstance, 'IMG3', 'JPEG');
15       try
16         Res.SaveToStream(MemStream);
17         MemStream.SaveToFile(ExtractFileName(ParamStr(0)+'\Img3.jpg'));
18         Img_Exibir.Picture.LoadFromFile(
           ExtractFileName(ParamStr(0)+'\Img3.jpg'));
19       finally
20         Res.Free;
21         MemStream.Free;
22         FreeResource(hFind);
23         FreeResource(hres);
24       end;
25     end
26     else
27       Application.MessageBox('Não Consegui Carregar o Arquivo',
                    'Atenção', MB_OK + MB_ICONASTERISK);
28   end
29   else
30     Application.MessageBox('Não Consegui Encontrar o Arquivo',
31                  'Atenção', MB_OK + MB_ICONASTERISK);
32 end;
```

Entre as linhas 03 e 05 temos a definição das variáveis. Na linha 07 usamos o método FindResource que verifica se existe o Resource e o conteúdo, caso haja, o retorno é atribuído a hFind. Na linha 10 o Resource é carregado e atribuído a hRes. Em seguida, é criado o MemoryStream e um ResourceStream (linha 14) que serão utilizados para a exibição da imagem JPEG. Na linha 16 o Resource salva o conteúdo dentro do MemoryStream, a seguir, na linha 17 é exportada a imagem, para ser carregada em seguida (linha 18). Entre as linhas 20 a 23 as variáveis são destruídas e liberadas da memória. Entre as linhas 26 a 30 temos as mensagens de erro em situações que o conteúdo não é carregado corretamente.

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Continuando, implementaremos o evento OnClick do Btn_Icone que é o próximo da sequência da Interface da aplicação. No próximo evento carregaremos um Cursor do mouse, para isso, implementamos o método da **Listagem 10**.

Na linha 5 é executado o método LoadCursor, atribuindo à Screen.Cursors, onde teremos a modificação do cursor do mouse relativo à posição da tela. Caso haja um Cursor válido, ele é zerado e vice-versa.

Continuando, o próximo evento será a implementação do OnClick referente ao botão Btn_DocTxt. Sua implementação encontra-se na **Listagem 11**.

Listagem 10. Evento OnClick do botão Btn_Cursor

```
01 procedure TFrm_Principal.Btn_CursorClick(Sender: TObject);
02 const
03  CursorRes = 1;
04 begin
05  Screen.Cursors[CursorRes] := LoadCursor(HInstance, 'CURSR');
06  if (Img_Exibir.Cursor = CursorRes) then
07  begin
08   Img_Exibir.Cursor := 0;
09   Btn_Cursor.Cursor := 0;
10  end
11  else
12  begin
13   Img_Exibir.Cursor := CursorRes;
14   Btn_Cursor.Cursor := CursorRes;
15  end;
16 end;
```

Listagem 11. Evento OnClcik do botão Btn_DocTxt

```
01 procedure TFrm_Principal.Btn_DocTxtClick(Sender: TObject);
02 var
03  Res: TResourceStream;
04  hFind, hRes: THandle;
05  Txt: TStringStream;
06 begin
07  hFind := FindResource(HInstance, 'DOCTXT', 'FILE');
08  if (hFind <> 0) then
09  begin
10   hRes := LoadResource(HInstance, hFind);
11   if (hRes <> 0) then
12   begin
13    Txt := TStringStream.Create;
14    Res := TResourceStream.Create(HInstance, 'DOCTXT', 'FILE');
15    try
16     Res.SaveToStream(Txt);
17     Mmo_txt.Lines.Text := Txt.DataString;
18    finally
19     Res.Free;
20     Txt.Free;
21     FreeResource(hFind);
22     FreeResource(hres);
23    end;
24   end
25   else
26    Application.MessageBox('Não Consegui Carregar o Arquivo',
                  'Atenção', MB_OK + MB_ICONASTERISK);
27  end
28  else
29   Application.MessageBox('Não Consegui Encontrar o Arquivo',
                'Atenção', MB_OK + MB_ICONASTERISK);
30 end;
```

Entre as linhas 03 a 05 temos a definição das variáveis. Na linha 07 é executado o método FindResource buscando o conteúdo de acordo com a Tag específica, atribuindo o resultado à variável hFind. Na linha 10 o recurso é carregado e atribuído à hRes. A Seguir, na linha 13 é criado um StringStream e um ResourceStream (linha 14). O método SaveToStream é executado na linha 16, salvando o conteúdo dentro do StringStream criado anteriormente. O seu conteúdo é então atribuído ao Memo (linha 17), por fim, temos a destruição e liberação dos recursos e objetos da memória (linhas 19 a 22). Entre as linhas 26 a 29 temos as validações referentes aos casos em que o resource não é carregado.

A seguir, o próximo botão é o responsável por executar o arquivo Wave, sendo assim, temos a chamada do método PlaySound dentro do evento OnClick do botão Btn_Musica, conforme **Listagem 12**.

Observe no código anterior que é utilizada a TAG do Resource ('MUSICA'), especificada no primeiro parâmetro, note como é simples a chamada do botão.

Para finalizar as implementações da aplicação principal, temos o código da **Listagem 13** que trata a execução de outro aplicativo.

Listagem 12. Implementação do evento OnClick do botão Btn_Musica

```
01 procedure TFrm_Principal.Btn_MusicaClick(Sender: TObject);
02 begin
03  PlaySound('MUSICA', HInstance, SND_ASYNC or SND_RESOURCE);
04 end;
```

Listagem 13. Implementação do evento OnClick do botão Btn_Programa

```
01 procedure TFrm_Principal.Btn_ProgramaClick(Sender: TObject);
02 var
03  Arquivo: String;
04  Res: TResourceStream;
05  hFind, hRes: THandle;
06 begin
07  Arquivo := ExtractFileName(ParamStr(0)+'\PrgSecundario.exe');
08  if not FileExists(Arquivo) then
09  begin
10   hFind := FindResource(HInstance, 'PROGRAMA', 'FILE');
11   if (hFind <> 0) then
12   begin
13    hRes := LoadResource(HInstance, hFind);
14    if (hRes <> 0) then
15    begin
16     Res := TResourceStream.Create(HInstance, 'PROGRAMA', 'FILE');
17     try
18      Res.SavetoFile(Arquivo);
19     finally
20      Res.Free;
21      FreeResource(hFind);
22      FreeResource(hres);
23     end;
24    end
25    else
26     Application.MessageBox('Não Consegui Carregar o Arquivo',
                  'Atenção', MB_OK + MB_ICONASTERISK);
27   end
28   else
29    Application.MessageBox('Não Consegui Encontrar o Arquivo',
                'Atenção', MB_OK + MB_ICONASTERISK);
30  end;
31  WinExecAndWait(Pchar(Arquivo), SW_SHOWNORMAL, True);
32 end;
```

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Entre as linhas 03 a 05 temos as variáveis utilizadas. Na linha 07 é utilizado o método ExtractFileName considerando o caminho do executável principal. O nome do arquivo é atribuído à variável Arquivo. Posteriormente, na linha 08 validamos se o caminho do arquivo é valido. Nas linhas 10 e 11 verificamos a existência do Resource, carregando-o em seguida na linha 13. A seguir, na linha 16 o ResourceStream é criado e tem seu conteúdo salvo na variável que armazena o caminho (arquivo - linha 18). Entre as linhas 20 a 22 temos a destruição e liberação dos recursos e, nas linhas 26 e 29 as mensagens de restrição relacionadas aos casos em que o recurso não foi carregado com sucesso. Com isso finalizamos as implementações dos botões.

Finalizando a aplicação de exemplo

Para finalizarmos a aplicação, a última modificação refere-se à adequação do Resource ao projeto, bastando simplesmente declarar o Resource na aplicação. Para isso, podemos utilizar a opção Project>View Source dentro do Delphi. Veja o código completo na **Listagem 14**. Repare que a linha 07 que inclui o recurso na aplicação através da diretiva {$R ArtResource}.

Listagem 14. Opção Project>View da aplicação

```
01 program PrgResource;
02 uses
03   Forms,
04   Unt_Principal in 'Unt_Principal.pas' {Frm_Principal},
05   Unt_Traduzir in 'Unt_Traduzir.pas';
06 {$R *.res}
07 {$R ArtResource}
08 begin
09   Application.Initialize;
10   Application.MainFormOnTaskbar := True;
11   Application.Title := 'Uso de Resource';
12   Application.CreateForm(TFrm_Principal, Frm_Principal);
13   Application.Run;
14 end.
```

As **Figuras 4** e **5** exibem respectivamente dois exemplos de execução do nosso exemplo, o primeiro é utilizando o recurso de línguas, já o segundo, mostra o carregamento do ícone.

Simplificar e agilizar a distribuição e atualização de um software é essencial, e através do uso de recursos, arquivos secundários requeridos pela aplicação podem ser incorporados ao aplicado, nada mais de "esquecer uma DLL", deixe o aplicativo principal lidar com isso.

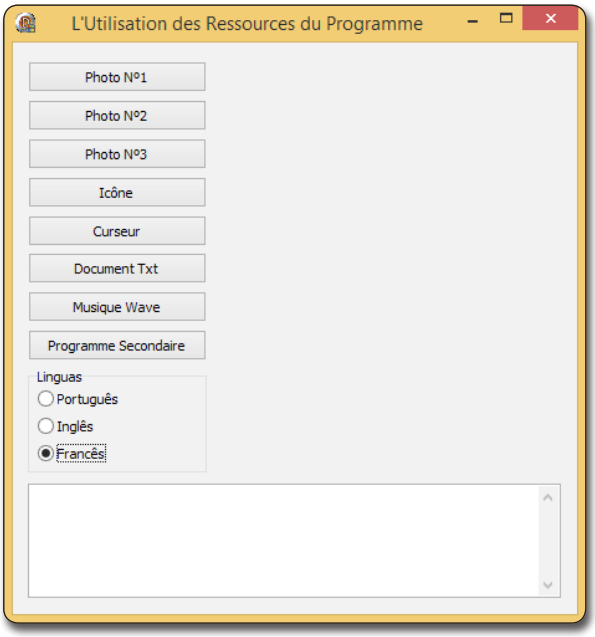

Figura 4. Utilização do recurso de línguas

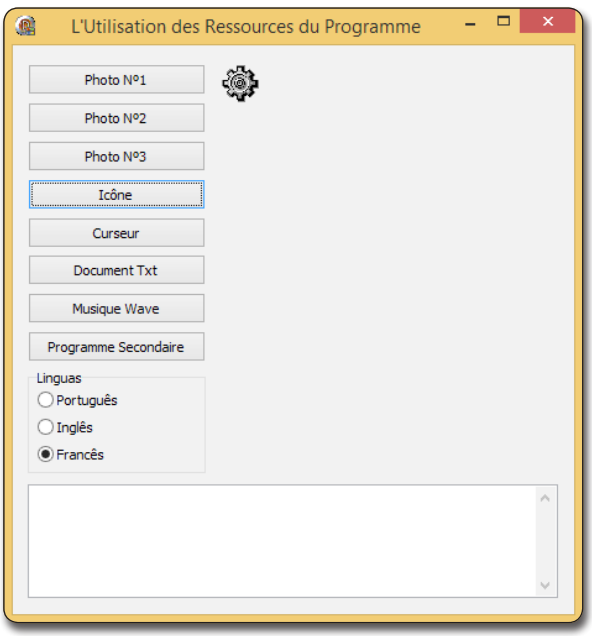

Figura 5. Carregamento do ícone

Autor

Vanderson Cavalcante de Freitas
vanderson.freitas@ig.com.br
Analista Desenvolvedor Delphi há mais de 5 anos, com experiência em médias e grandes empresas de São Paulo. Formado em técnico em informática no ano de 2003, com diversos cursos em formação específica, como Oracle, Delphi e C#.

Você gostou deste artigo?

Dê seu voto em ***www.devmedia.com.br/clubedelphi/feedback***

Ajude-nos a manter a qualidade da revista!

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Sistema de créditos DevMedia

Conteúdo **completo** ao seu alcance

Com o sistema de créditos DevMedia você pode comprar individualmente mais de 4.000 vídeo aulas, 8.000 artigos, além de palestras e vídeos com formações completas.

TODAS as vídeo aulas que você sempre quis ver a partir de

R$ 1,50

ECONOMIA: você não precisa assinar 8 produtos diferentes para ter acesso ao conteúdo completo do site!

Saiba mais sobre o Sistema de Créditos!

www.devmedia.com.br/creditos

DEVMEDIA

Somos tão apaixonados por tecnologia que o nome da empresa

diz tudo.

Porta 80 é o melhor que a Internet pode oferecer para sua empresa.

Já completamos 8 anos e estamos a caminho dos 80, junto com nossos clientes.

Adoramos tecnologia.
Somos uma equipe composta de gente que entende e gosta do que faz,

assim como você.

Estrutura

100% NACIONAL. Servidores de primeira linha, links de alta capacidade.

Suporte diferenciado

Treinamos nossa equipe para fazer mais e melhor. Muito além do esperado.

Serviços

Oferecemos a tecnologia mais moderna, serviços diferenciados e antenados com as suas necessidades.

1-to-1

Conhecemos nossos clientes. Atendemos cada necessidade de forma única.
Conheça!

Porta 80
WEB HOSTING

Hospedagem | Cloud Computing | Dedicados | VoIP | Ecommerce | Aplicações | Streaming | Email corporativo

porta80.com.br | comercial@porta80.com.br | twitter.com/porta80

SP 4063-8616 | RJ 4063-5092 | MG 4063-8120 | DF 4063-7486